RL-TR-96-78
Final Technical Report
August 1996

# TRUSTED DISTRIBUTED RUBIX

Infosystems Technology, Inc.

Mohammed Hasan, James P. O'Connor, Greg Pryzby, Anath Rao, Jasbir Singh, Mark Smith, and Dr. Virgil Gligor

Rome Laboratory
Air Force Materiel Command
Rome, New York

RL-TR-96-68 has been reviewed and is approved for publication.

APPROVED: Mary L. Denz
MARY L. DENZ
Project Engineer

FOR THE COMMANDER: John A. Graniero
JOHN A. GRANIERO
Chief Scientist
Command, Control & Communications Directorate

# REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

| | | |
|---|---|---|
| 1. AGENCY USE ONLY (Leave Blank) | 2. REPORT DATE<br>August 1996 | 3. REPORT TYPE AND DATES COVERED<br>Final Sep 93 - Sep 94 |

| | |
|---|---|
| 4. TITLE AND SUBTITLE<br>TRUSTED DISTRIBUTED RUBIX | 5. FUNDING NUMBERS<br>C - F30602-94-C-0132<br>PE - 33140F<br>PR - 7820<br>TA - 04<br>WU - 25 |
| 6. AUTHOR(S)<br>Mohammed Hasan, James P. O'Connor, Greg Pryzby, Ananth Rao, Jasbir Singh, Mark Smith, and Dr. Virgil Gligor | |
| 7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)<br>Infosystems Technology, Inc.<br>6411 Ivy Lane, Suite 306<br>Greenbelt MD 20770 | 8. PERFORMING ORGANIZATION REPORT NUMBER<br>N/A |
| 9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)<br>Rome Laboratory (C3AB)<br>525 Brooks Rd<br>Rome NY 13441-4505 | 10. SPONSORING/MONITORING AGENCY REPORT NUMBER<br>RL-TR-96-78 |

11. SUPPLEMENTARY NOTES

Rome Laboratory Project Engineer: Mary L. Denz/C3AB/(315)330-3241

| | |
|---|---|
| 12a. DISTRIBUTION/AVAILABILITY STATEMENT<br>Approved for public release; distribution unlimited | 12b. DISTRIBUTION CODE |

13. ABSTRACT (Maximum 200 words)

This report presents the results of the Trusted Distributed RUBIX project. This project was a research effort to address the practical issues of developing high-assurance multilevel secure distributed database management systems. The goal of this effort was to advance the state-of-the-art in multilevel secure distributed database management in order to shorten the time before open systems/distributed computing solutions are available to user communities having multilevel security requirements.

| | |
|---|---|
| 14. SUBJECT TERMS<br>Trusted distributed database management system (DBMS), Multilevel secure, Relational DBMS | 15. NUMBER OF PAGES<br>68 |
| | 16. PRICE CODE |

| | | | |
|---|---|---|---|
| 17. SECURITY CLASSIFICATION OF REPORT<br>UNCLASSIFIED | 18. SECURITY CLASSIFICATION OF THIS PAGE<br>UNCLASSIFIED | 19. SECURITY CLASSIFICATION OF ABSTRACT<br>UNCLASSIFIED | 20. LIMITATION OF ABSTRACT<br>SAR |

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18
298-102

## Acknowledgment

Significant contributions to this work were made by Mohammed Hasan, James P. O'Connor, Greg Pryzby, Ananth Rao, Jasbir Singh and Mark Smith.
Dr Virgil Gligor wrote the Model Composition section, appearing as Appendix A.
Infosystems Technology Inc., would like to thank them for their effort.

# Contents

## LIST OF FIGURES

# 1 Introduction

In many DoD and Intelligence organizations, large databases exist which are distributed on mainframe computers, workstations, and personal computers. Users in these organizations need shared access to these multiple autonomous databases, however they must sacrifice efficient open systems/distributed computing solutions in order to meet security requirements. As a result, users must adopt solutions where distributed data are isolated on system high computers and networks, and the flow of information between systems is controlled through the use of guard technology. Such solutions severely limit the user's ability to access and consolidate data from distributed databases (logistics, command and control, intelligence, etc) in a timely manner. This inability to obtain mission critical information in a timely manner can result in a compromise of mission objectives.

This document presents the results of the Trusted Distributed RUBIX project. This project was a research effort to address the practical issues of developing high-assurance multilevel secure distributed database management systems. The goal of this effort was to advance the state-of-the-art in multilevel secure distributed database management in order to shorten the time before open systems/distributed computing solutions are available to user communities having multilevel security requirements. The issues identified for investigation over the course of this effort were:

1. Distributed transaction management in a multilevel environment;

2. Distributed query processing in a multilevel environment;

3. Interactions between open systems standards and security;

4. Security Policies for multilevel secure (MLS) distributed database management system (DDBMS); and

5. Infrastructure requirements for MLS D-DBMS.

The approach that we followed in this effort was to investigate these issues by developing a prototype multilevel secure distributed DBMS. The prototype was developed by extending Trusted RUBIX. Trusted RUBIX is a high assurance multilevel secure client/server relational DBMS [1, 2, 3, 4, 5]. The requirements and design for the Trusted Distributed RUBIX system are specified in [6, 7, 8, 9].

## 1.1 Report Structure

Section 2 of this report presents background information. Included in this section is a description of Trusted RUBIX, an overview or distributed transaction processing and distributed query processing, and a discussion of the issues involved in providing these capabilities in an MLS environment. Section 3 presents an overview of the Trusted Distributed RUBIX architecture including its security characteristics and the strengths and weaknesses of the architecture. Section 4 discusses the Trusted Distributed RUBIX security policy.

Section 5 discusses improvements to the Trusted RUBIX Server that were made as a part of this effort. Section 6 presents conclusions, and section 7 discusses future work.

## 2 Background

### 2.1 Trusted RUBIX

The approach followed in this effort was to investigate the issues of multilevel secure data management by developing a prototype multilevel secure distributed DBMS. The prototype was developed by extending Trusted RUBIX MLS relational DBMS. This section provides an overview of the Trusted RUBIX architecture.

#### 2.1.1 Server Architecture

The Trusted RUBIX server architecture is based on the concept of TCB subsetting [10, 11]. This section introduces the concept of TCB subsetting and provides an overview of the Trusted RUBIX TCB architecture.

**2.1.1.1 TCB Subsets** The Trusted RUBIX TCB architecture is based on the concept of TCB subsetting. As defined in the Trusted Database Interpretation (TDI) [11] of the Trusted Computer System Evaluation Criteria (TCSEC) [12], a TCB subset M is a set of software, firmware, and hardware (where any of these three could be absent) that mediates the access of a set S of subjects to a set O of objects on the basis of a stated access control policy P and satisfies the properties:

- M mediates every access to objects in O by subjects in S;
- M is tamper resistant; and
- M is small enough to be subject to analysis and tests, the completeness of which can be assured.

Based on this definition, a system can be composed of multiple hierarchically related subsets. These subsets are related by the relation "depends-upon". A subset that does not depend upon any other subsets is called a primitive subset. Each subset M (except for the most primitive subsets) uses the resources of the more primitive subsets to enforce its security policy.

**2.1.1.2 Trusted RUBIX Subset Architecture** The Trusted RUBIX TCB architecture is shown in Figure 1 and consists of two subsets: the Trusted RUBIX Kernel (M[1]), which is responsible for implementing a mandatory security policy, and the SQL Engine (M[2]) which is responsible for implementing the labeling, consistency, and discretionary access control policy. The M[1] subset consists of the operating system TCB along with the minimal amount of trusted code required to implement the desired DBMS policy, thus

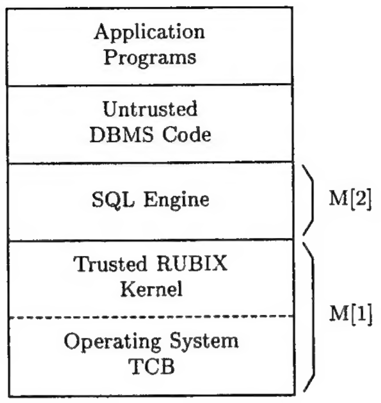

Figure 1: Trusted RUBIX Subset Architecture

running as a trusted subject on the underlying operating system. The M[2] subset is layered on the M[1] subset and implements a policy that is a further restriction of the policy enforced by the M[1] subset. The M[2] subset is completely constrained by the policy of the M[1] subset.

**2.1.1.3 Trusted RUBIX Process Architecture** The Trusted RUBIX Kernel and SQL Engine are implemented as two separate operating system processes (Figure 2). Conceptually, the interface to each of these processes is a set of procedures. These interfaces are the TCB interfaces for the two subsets. From a client's point of view, all that is necessary to access the services of a subset is to link to the appropriate library containing these procedures.

These interface procedures are implemented as a form of (local) remote procedure call. There are two versions of each procedure: a client-side procedure that is linked into a non-TCB process and a server-side procedure that runs in the TCB domain. Each client-side call to an interface function causes a peer function to be executed on the server. When an initialization routine is called, a UNIX pipe is established between the non-TCB process and the server process for synchronous communication. When the non-TCB program calls one of the client-side access functions, the arguments to the call are collected and passed, along with a procedure identifier to the server side. The client procedure then blocks, and waits for a response. On the server side, a dispatcher function examines the procedure identifier and calls the appropriate server-side procedure with the communicated arguments. When this call returns, the dispatcher function sends back any return values to the client procedure which then returns like a normal procedure call. The dispatcher function then blocks, awaiting the next request. The communication channel is brought down by a termination routine.

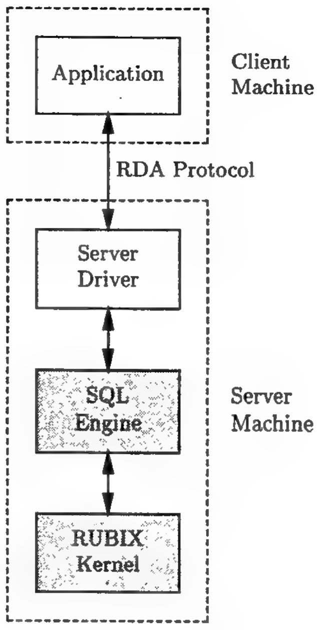

Figure 2: Trusted RUBIX Process Architecture

### 2.1.2 Trusted RUBIX Client/Server Architecture

The TD-RUBIX prototype will be developed by extending Trusted RUBIX. Trusted RUBIX is an SQL-based client-server MLS Relational DBMS designed to meet the TCSEC criteria for a class B2 computer system. The Trusted RUBIX client-server architecture is shown in Figure 3. Clients are untrusted application programs that have been linked with the Trusted RUBIX client software so that they can communicate with the Trusted RUBIX server. There is one instantiation of the server for each active client. A given client and server pair can run on the same machine or on different machines connected via a network. The server must reside on the same machine as the data to be accessed. All communication between client and server takes place on a single-level connection, using the SQL Remote Database Access (RDA) standard protocol. A client can access multiple servers concurrently (although no mechanism for distributed transaction management is currently provided). Currently, three types of clients are supported: instantiations of an Interactive SQL (ISQL) interface that provides ad hoc access to databases, user-developed Embedded SQL (ESQL) applications, and user-developed applications utilizing the SQL Call-Level Interface (CLI) [13].

The current version of Trusted RUBIX is implemented above the secure networking facilities of UNIX System V Release 4.2 ES. Trusted RUBIX uses these services in such a way that no trusted code is introduced to support client-server operation. These facilities can be replaced with networking facilities meeting the following requirements:

- *Remote Process Invocation.* Provide a way for a client to invoke a RUBIX server process on a remote machine at the invoker's current level, user-id, and group-id.

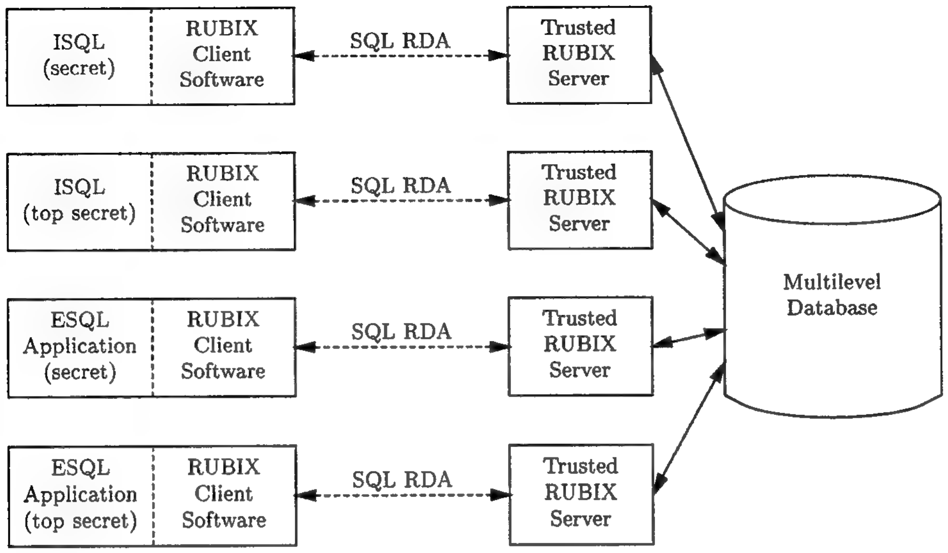

Figure 3: Trusted RUBIX Client-Server Architecture.

- *Single-level connection establishment.* Provide a way to establish a single-level connection between the invoker and the new process.
- *Network security policy enforcement.* Ensure that the services do not violate intra- or inter-host mandatory or discretionary access controls.
- *Data Confidentiality.* Protect the confidentiality of transmitted data (this can be accomplished by physically protecting the transmission media).
- *Attribute Mapping.* Perform user-id, group-id, and security level mapping.

The infrastructure we selected for the implementation was the secure networking services of UNIX SV4.2ES. The primary components of the UNIX SV4.2ES secure networking facilities are the connection server and listen port monitor (Figure 4). The connection server handles all client-side connection establishment as a single service. The listen port monitor is a daemon that listens on the server machine for incoming connection requests, accepts the requests, and starts services that have been requested. In order for these facilities to be used to provide a service (e.g., Trusted RUBIX), the service must have been previously registered with the connection server on the client machine and the listen port monitor on the server machine.

When an application needs to access a service on a remote machine, it makes a request to the connection server. The connection server validates that the connection to the remote machine is permitted at the application's level. If the connection is permitted, the connection server opens a connection to the corresponding listen port monitor on the remote machine. Once the connection is established, the connection server and listen port monitor exercise a mutual authentication scheme (a cryptographic scheme based on secret keys).

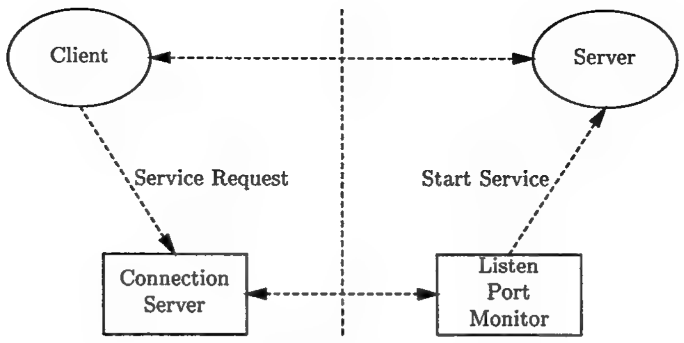

Figure 4: UNIX SV4.2ES Secure Networking Facilities

If the authentication succeeds, the connection server passes the client's user-id, group-id, and security level to the port monitor. The port monitor maps these attributes to local attributes (if necessary) and starts the server corresponding to the requested service with the appropriate user attributes. The connection server then passes its end of the connection to the client, and the port monitor passes its end of the connection to the newly invoked server. The client and server are now connected at the desired security level and can begin a session. At this point there is a server process operating on behalf of the user on the remote machine which is running with the user's credentials (possibly mapped).

Two characteristics of the Trusted RUBIX architecture make it particularly well-suited for this effort. First, it supports client-server operation without the introduction of any trusted code. Such a system can have a high-level of assurance because it derives its assurance directly from the underlying secure distributed computing infrastructure. Second, all communication between the client and server portions of Trusted RUBIX is accomplished via the SQL RDA protocol. This makes this architecture suitable for a high-assurance open-systems based distributed DBMS.

### 2.1.3 Multiversion Timestamp Ordering

Trusted RUBIX uses Multi-Version Timestamp Ordering (MVTO) as a concurrency control mechanism. The original Trusted RUBIX MVTO algorithm was non-secure. The non-secure algorithm worked as follows. A transaction is assigned a unique timestamp when it begins executing, and this timestamp is associated with all read and write operations of the transaction. Each tuple can have multiple versions, and each version has an *in*, *out*, and *read* timestamp associated with it. When a tuple is written, a new version of the tuple is created and its *in* timestamp is set to the timestamp of the current transaction. If a previous version of the tuple existed, its *out* timestamp is set to the timestamp of the current transaction. When a tuple is read, the version of the tuple with an *in* timestamp that is less than the timestamp of the current transaction and an *out* timestamp that is greater than the timestamp of the current transaction is retrieved. The tuple's *read* stamp is set to the timestamp of the current transaction. An exception is raised (and the transaction aborted) if an update is attempted on a tuple that has been read by a later transaction, or if an insertion is attempted within a relation that has been read by a later transaction.

This mechanism ensures that actions of multiple transactions will not conflict and the results of their actions will be as if they had executed sequentially in order of their timestamps. That is, their execution will be serializable. The problem with this algorithm is that it allows a low level transaction to be rolled back by the action of a high level transaction (viz., a read). This represents a covert channel. To eliminate this channel, the Trusted RUBIX algorithm was modified to be conflict-secure. The new algorithm is a variant of the algorithm proposed in [14]. In the non-secure algorithm, when a transaction starts, it is assigned the current time as its timestamp. In the secure algorithm, a transaction is assigned a timestamp that proceeds the timestamps of all active transactions that it strictly dominates. This effectively moves the transaction to a time in the past where it cannot possibly conflict with a low transaction. The modified algorithm eliminates the above channel and produces serializable executions. The cost associated with this approach is that a high transaction may see data that is less current because its execution has effectively been pushed into the past.

The key to the above algorithm is the selection of a transaction's timestamp. The timestamp is selected by searching through the list of currently active transactions and finding the earliest transaction that is strictly dominated by the current transaction. This transaction is called the earliest dominated transaction (EDT). If an EDT does not exist, then the transaction timestamp is set to the current time. If an EDT does exist, then a timestamp is selected that is between the timestamp of the EDT and that of the previous transaction. If no previous transaction exists, the timestamp selected is the one between the EDT and the latest consistent moment (LCM). The LCM is a point in the database history before which the state of the database is stable (i.e., no active transactions exist with a timestamp prior to the LCM). It would be sufficient to select a transaction time that was one microsecond prior to the time of the EDT, but the above approach was adopted to leave room for future transactions following the same algorithm.

## 2.2 Distributed Transaction Processing

### 2.2.1 Two-Phase Commit Protocols

A two-phase commit protocol is a protocol that allows a set of sites participating in a distributed transaction to cooperate in such a way that the atomic execution of the transaction is assured [15]. The two-phase commit protocol guarantees that all participants in a distributed transaction "go the same way", that is, they either all commit the transaction, or they all roll it back. Furthermore, this property is guaranteed even in the case of network or site failure. The entities in the two-phase commit protocol are a single transaction manager and two or more resource managers.

In the first phase of the protocol, the transaction manager sends each of the participants a prepare-to-commit message which informs them to enter a state where they can "go either way", that is, either commit or rollback. To do this, the resource managers must force their transaction logs to disk so that the transaction is recoverable. Each transaction manager then sends the coordinator a message indicating the success or failure of the prepare.

Based on the responses of the resource managers, the transaction manager decides which way the transaction should go. If all of the responses are "success", then the decision will be to commit. If any of the responses is "failure", then the decision will be to rollback. The transaction manager then records the decision in a global transaction log and informs each of the resource managers to either commit or rollback. The protocol requires the resource managers to abide by the transaction manager's decision.

The recording of the transaction managers decision in the transaction log marks the beginning of the second phase. If there is a failure in any portion of the process, the transaction manager can recover the transaction by checking the disposition of the global transaction in the global transaction log and then informing the participants of the decision when they become available. If there is no decision record in the global transaction log, then the protocol proceeds as if a rollback decision was found.

### 2.2.2 X/OPEN DTP Model

X/Open has published a reference model for distributed transaction processing [16] along with an associated set of standards. The X/Open distributed transaction processing (DTP) model, show in figure 5, consists of three communicating entities: the application program (AP), the resource manager (RM), and the transaction manager (TM). The AP defines the boundaries of a transaction and specifies the actions that constitute a transaction. The AP interfaces with a TM through the X/Open TX interface [17] and with RMs through an RM-supported interface (e.g., SQL CLI). The TM manages global transactions and transaction recovery. The RM generates global transaction identifiers, coordinates the two-phase commit process, and recovers transactions using a global transaction log. The TM communicates with the RM using the XA interface [18]. Finally, the RM provides shared access to some set of resources. The RM is capable of being a participant in the two-phase commit process.

### 2.2.3 Security Issues

There are a number of issues that must be addressed when considering supporting atomic transactions in a multilevel environment. The first issue is whether the transactions themselves are to be single-level or multilevel. A single-level transaction is a transaction in which the AP connects to each of the resource managers at the same security level. Single level transactions have the advantage that, since the participants are all at the same level, there are no covert channels in the two-phase commit protocol itself. If the underlying RMs are multilevel, then two other issues can arise. The first issue is that the secure concurrency control mechanism of the underlying RM may not be compatible with the two-phase commit protocol [19]. The second issue is that covert channels may be introduced into the RM as part of the functions that support its participation in a distributed transaction.

A multilevel transaction is a transaction where an AP can connect to RMs at different security levels [20]. In order to connect to RMs at different levels the AP must be a trusted

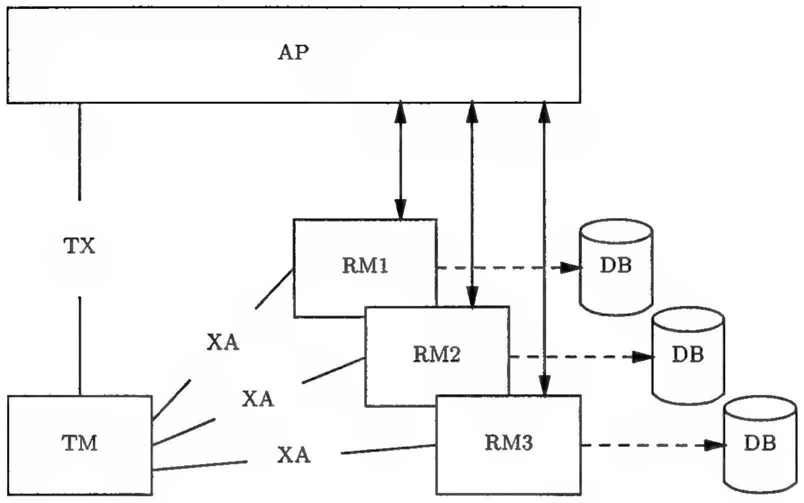

Figure 5: X/OPEN Distributed Transaction Processing Model.

process. In the case of multilevel transactions, the protocol itself includes covert channels. To see that this is the case, consider a transaction that covers two levels, high and low, in this case if the high RM fails the prepare, then the low transaction will be forced to abort. The results is an information flow from high to low.

A final issue that must be considered in supporting distributed transactions is the trust characteristics of the components. In the multilevel transaction case, both the AP and the TM will require some degree of trust since they will be communicating with RMs over connections at different levels. In the single-level transaction case, there is the possibility that these components can be untrusted. If the responsibility for security policy enforcement is distributed across system components, then it will be much more difficult to attain a high-level of assurance that the security policy is enforced correctly. This is a result of the complexity inherent in distributed systems. However, if the responsibility for security policy enforcement is centralized, the assurance argument is simplified considerably.

## 2.3 Distributed Query Processing

Distributed query processing supports the execution of queries that access data that are distributed across multiple sites on a network. A distributed query processing capability allows data to be placed close to the users that access it most frequently, but also allow users to submit queries to access the distributed data as if it were stored in their local DBMS.

The distributed query processing approach that we are focusing on in this effort is known as a loosely-coupled federated approach. In this approach, users are provided with integrated access to multiple independent autonomous DBMSs. This approach is characterized by the fact that the schema for the individual DBMSs are developed and maintained independently (i.e., there is no one single global conceptual schema for the entire distributed DBMS). In this effort, we investigated approaches with and without federated schemata. If

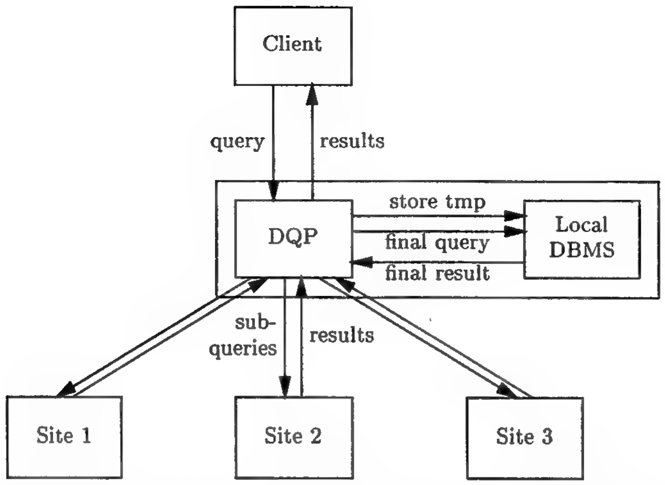

Figure 6: Distributed Query Processing Reference Model.

there is no federated schemata, users can submit queries that access distributed data, but they must know the location of the data and explicitly place that information in the query. If there is a federated schema, the federated schema can provide the user with an integrated view of distributed data that makes the distribution of data transparent.

### 2.3.1 DQP Reference Model

A generic architecture for loosely-coupled federated D-DBMS is shown in figure 6. In this model a client submits queries to the local site. The local site checks if the query references non-local data, and if it does, the distributed query processor will generate a set of subqueries to access the non-local data. The distributed query processor constructs these subqueries in such a way as to minimize the amount of data that must be transferred across the network. This is done by sending the most selective subquery possible to the remote site. The subqueries are executed at the remote sites and the results are retrieved and stored in temporary tables in the local DBMS. A final query is then executed on these partial results to produce the final result which is returned to the requesting user.

### 2.3.2 Security Issues

There are a number of issues that must be addressed in the area of secure distributed query processing. The first set of issues arise because you now have a DBMS accessing other DBMSs as the result of a end-user request. The question that arises is who is the principal associated with remote site access. Does the distributed query processor access the remote site as the end-user or as a trusted peer? If the DQP accesses the remote site as the end-user, how is the indirect authentication performed? Does the user need to divulge authentication data to the distributed query processor? How is a trusted path guaranteed? If the distributed DBMS accesses the remote DBMS as a trusted peer, how does the remote

DBMS enforce its security policy? Does it delegate access control responsibility to the local DBMS? What does this do to the autonomy of the remote DBMS? What is the result on the assurance characteristics of the entire system?

A closely related issue is the issue of federated policy. Since a distributed query processing capability allows the user to aggregate data from many sources, it is reasonable to implement an access control policy for the federated data that is in addition to the access control policies of the individual DBMSs. This requires that a portion of the DQP be trusted because it now enforces policy. An additional issue that is raised is how the federated policy relates to the local DBMS policies. If it is a further restriction of the local policies, what is to prevent the policy from being circumvented by going directly to the data sources. If it is a relaxation of the local policies, there must be a trust relationship between the distributed query processor and the local DBMSs.

Another set of issues arises from the introduction of multilevel security into the system. The first is the impact of different system accreditation ranges on query processing. For example, consider the situation where one system A is accredited to handle U through S, and system B is accredited to handle S through TS. If a TS user on system A submits a query that needs to access data on system B, the resulting subquery cannot be sent to the lower level system without introducing a covert channel.

A second issue related to multilevel security involves how to reliably communicate labeled information. This issue must be considered in light of the fact that the labels returned by most DBMSs are inherently advisory and are therefore unreliable as an indicator of the true sensitivity of the data [21]. This suggests one solution which is to accept that the labels are advisory in nature and not place any more concern in protecting their integrity than you would for any other data. If you assume that labels are reliable, and you want to preserve that reliability, the problem is how to pass labels from one DBMS to another in such a way that their reliability is maintained. There are actually two problems that must be addressed. The first is that the binding of the label to the tuple must be maintained during transmission. The second is that the same label may have different semantics on the source and destination sites. This second problem is a special case of the problem of semantic heterogeneity [22] and will not be considered further here. The first problem could be addressed by using cryptographic techniques to bind the label to the data. This approach would require trusted code on each side of the transfer to compute/validate checksums and to downgrade the tuple based on the attached label. A second approach is to process queries so that there is one subquery for each level of data to be returned. This eliminates the trusted code on the remote end but still requires the DQP to be trusted so that it can send out queries at different levels.

A related labeling issue is how distributed query processing impacts result labeling. Result overclassification is due to a combination of the fact that tuple level labels are inherently advisory and that the distributed query processor stores partial results in temporary relations. The query processor cannot store the results using the labels from the remote site because the labels are advisory (not reliable) and once stored in the local DBMS will be used as a basis for access control decisions. The alternative is to store them at the level of the connection, but then the partial result will be labeled at a different level than the corresponding data on the remote host. The result of this is that the result returned to the user will be labeled differently than if all the data had been stored on the local host.

A final issue is what are the trust characteristics of the distributed query processing software. Does it need to be trusted? Does it need to be multilevel? How much of it needs to be trusted? The answers to these questions depend largely on the resolution of the above issues in a particular system design.

## 3 Prototype Architecture

Trusted Distributed RUBIX extends the client/server version of Trusted RUBIX to support two forms of distributed database access. These capabilities provided in two separate prototype increments: the distributed access prototype and the distributed processing prototype.

The primary design goal of the distributed access prototype was to provide a single-level atomic multi-site transaction processing capability in a multilevel environment with a high level of security assurance. A secondary goal was to develop an architecture based on open systems standards so it would be possible to use a commercial transaction manager to structure multilevel transactions accessing heterogeneous resource manager types.

The primary design goal of the distributed processing prototype was to provide integrated access to distributed multilevel data (read only) with a high level of security assurance. A secondary goal was to provide a degree of location transparency to system users by allowing them to develop their own federated schemata. A final design goal was to develop an architecture that is extendible to read/write access of remote data and access to heterogeneous data sources.

### 3.1 Distributed Access Prototype

The distributed access prototype, shown in Figure 7, extends Trusted RUBIX to permit clients to execute atomic read/write transactions that access data on multiple servers. This capability is provided by extending the Trusted RUBIX client, server, and connectivity software. The security approach taken for this prototype is to follow the philosophy of the client/server version of Trusted RUBIX and support this capability without any trusted code in the client or connectivity software. The following sections describe the modifications that were made to Trusted RUBIX to support this capability.

#### 3.1.1 Client Modifications

The primary extension to the client software was the addition of an untrusted transaction manager to coordinate distributed transactions. This TM was based completely on the standards of the X/Open DTP model. In order for the TM to communicate distributed transaction management requests to the Trusted RUBIX server, support for the X/Open XA interface was added to the Trusted RUBIX client library. Finally, the Trusted RUBIX interface software (ISQL, ESQL, and SQL CLI) was modified to support a distributed transaction

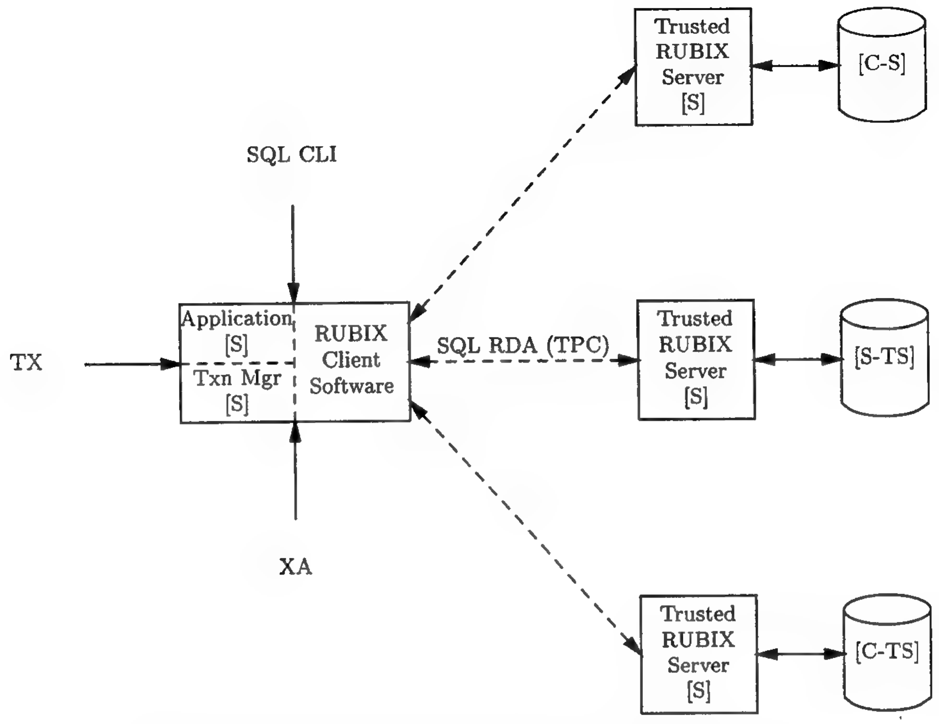

Figure 7: Distributed Access Prototype Architecture.

mode. In distributed transaction mode, these components demarcate distributed transactions by sending transaction requests to the TM via the TX interface. All of the software is untrusted. Only two modifications to this design were necessary due to multilevel security. The first was that there needed to be a global transaction log per security level (maintained as a file in a UNIX multilevel directory). This is because different instances of the transaction manager can run at different security levels, therefore they cannot read/write the same transaction log. The second allowance was the inclusion of the security level into the transaction identifier. This was necessary because the transaction managers at different levels do not share information and including the security level in the transaction identifier ensures that unique transaction identifiers are selected.

### 3.1.2 Server Modifications

The extensions to the server software were those necessary to support the 2-phase commit protocol. The changes were primarily: support for TM supplied transaction identifiers, support for the "prepared" transaction state, and support for global transaction recovery. Supporting these capabilities required modifications to the Trusted RUBIX kernel and SQL Engine. Because these changes modify the Trusted RUBIX TCB, their impact on the security properties of the system was carefully considered.

The impact of the first change was that, in the modified system, when users start a transaction on the RM, they specify a unique external transaction identifier that the system

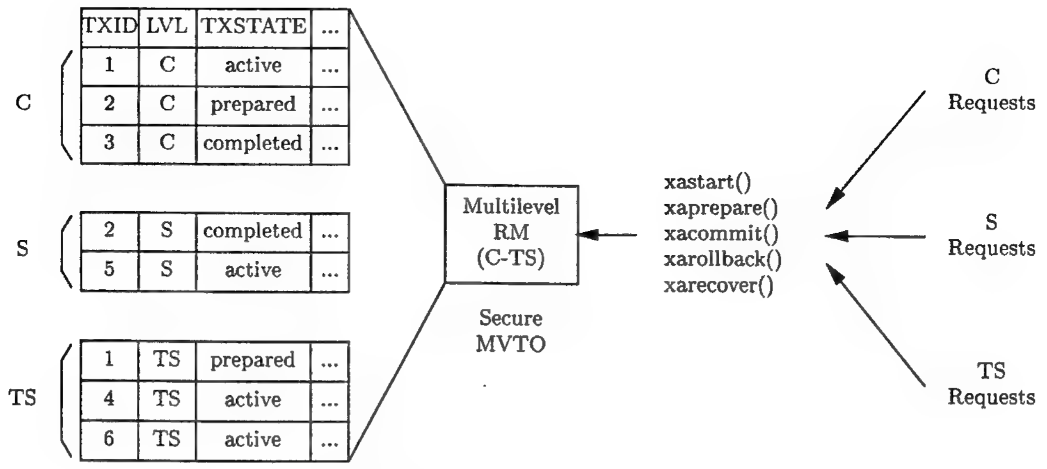

Figure 8: Multilevel Resource Manager.

is required to use to identify a transaction. The problem with such a transaction identifier is that it is possible that two users could select the same transaction identifiers. In this case, the system would have to resolve this conflict by returning an error to the second user who attempts to use that transaction identifier. This is not normally a problem, but in the case where the two users are at different levels, this can be used as a signaling channel. For example, if a high user wants to signal information to a low user, the high user need only start a transaction with a predetermined transaction identifier at a predetermined time. The low user can then test this by attempting to create a transaction with the same transaction identifier. The resulting conflict (or lack of conflict) will signal the desired information. The solution to this problem is to "polyinstantiate" transaction identifiers. That is, each transaction is uniquely identified by the user specified transaction identifier plus the level of the transaction. This means that two transactions can exist at different levels with the same user specified transaction identifiers, but they are different transactions because the system differentiates between them internally using the level of the transaction. Since the transaction space is now partitioned across security levels, the problem of conflicting transaction identifiers is eliminated. This partitioning of the transaction space is depicted in figure 8. A side effect of this partitioning is that operations on transactions only see transactions at a single level (i.e., when is comes to transactions, there is no read-up or read-down).

A second issue related to transaction identifiers is the question of who is authorized to operate on a transaction with a particular transaction identifier. The policy decision made for this prototype is based on the fact that the TM connects to the RM under a particular user identifier and at a particular level. The Trusted RUBIX server only allows a user to operate on a transaction that was invoked by the current user and that has a level equal to the user's current level.

The addition of the "prepared" state to transactions has security implications as well. The problem is that a prepared transaction is not complete and therefore can conflict with

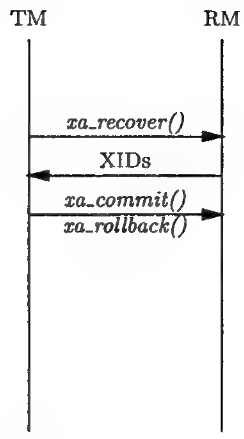

Figure 9: Distributed Transaction Recovery

other concurrent transactions. If these transactions are at a lower level such conflicts can be used as a signaling channel. As it turns out, this is not a problem. As discussed in section 2.1.3, the only way for a high transaction to conflict with a low transaction is to read down. Since a transaction cannot do any more work once it enters the prepared state, then there can be no additional conflicts caused by an extended period in the prepared state. The standard conflict-secure MVTO algorithm ensures that none of the reads done prior to the prepare cause any high-to-low conflicts.

The final change to the Trusted RUBIX server was support for transaction recovery. In order to perform distributed recovery after a failure, the TM must be able to obtain from each RM the set of transactions that have been prepared but not committed. The TM can then consult its global transaction log to determine the disposition of the uncommitted branches and inform the RM as to how the transaction should be completed. This process is accomplished using the *xa_recover()*, *xa_commit*, and *xa_rollback()* calls as shown in figure 9. The security concern here is that the policy for the return of transaction information is consistent with the existing mandatory or discretionary access control policies. The extensions that were made to the policy are based on the fact that the TM connects to the RM under a particular user identifier at a particular level. The Trusted RUBIX server only returns uncommitted transactions that were invoked by that user at the current level.

### 3.1.3 Connectivity Modifications

The extensions to the Trusted RUBIX connectivity software are those necessary to support distributed transaction processing. In the client/server version of Trusted RUBIX, communication between client and server was via the SQL Remote Data Access (RDA) protocol using the basic application context (BAC) [23, 24]. In the basic application context, the RDA transaction processing services are used to provide transaction processing on a per-connection basis. To support distributed transaction processing, the implementation of RDA was modified to support the RDA transaction processing context (TPC).

In the TPC, an application can have a single distributed transaction that spans multiple connections. To support the RDA TPC, it was necessary to implement the subset of the ISO TP services. The interface to these services was a subset of the XAP-TP interface [25] which is is an X/OPEN defined interface to the OSI TP services. It should be noted that the implementation supports the TP functionality (services), but not the TP protocol itself. This is because an implementation of the TP protocol was beyond the scope of this effort. Supporting the TP services, and basing the implementation on a standard interface, will allow migration to a commercial off-the-shelf (COTS) version of TP when one becomes available on the target platform.

### 3.1.4 Example

The user accesses the Trusted RUBIX distributed transaction processing capabilities through SQL. The user uses the CONNECT statement to connect to one or more sites. When a user submits a query, the query is executed in the context of the current connection. The user sets the current connection with the SET CONNECTION statement. A global transaction is initiated when the user issues a transaction initiating statement and no transaction exists. All subsequent statements are executed within the context of that global transaction. The user terminates a global transaction with the COMMIT or ROLLBACK statement. The COMMIT causes the TM to attempt to commit the transaction using the two-phase commit protocol. A ROLLBACK causes the TM to rollback the transaction.

The following example is an SQL program to transfer weapons between sites. Note that this example omits validation and error handling functions that would be part of an actual application.

```
transfer_weapons(site1, site2, weapon, quantity)
{
    /* ... */

    EXEC SQL CONNECT TO site1 AS c1;
    EXEC SQL CONNECT TO site2 AS c2;

    EXEC SQL SET CONNECTION C1;

    EXEC SQL UPDATE wing_weapons
             SET    amount = amount - :quantity;
             WHERE  weapon_id = :weapon

    EXEC SQL SET CONNECTION C2;

    EXEC SQL UPDATE wing_weapons
             SET    amount = amount + :quantity;
             WHERE  weapon_id = :weapon
```

```
        EXEC SQL COMMIT;

        /* ... */
    }
```

The transfer is structured as a single global transaction involving the sites passed in as arguments: *site1* and *site2*. The program connects to each of the sites with the SQL CONNECT statement. The current connection is set to *site1* at which point the amount of *weapon* in the wing_weapons table at *site1* is debited by *quantity*. Since this is the first executable statement, this statement causes a global transaction to be initiated. The current connection is then set to *site2* at which point the amount of *weapon* in the wing_weapons table at *site2* is credited by *quantity*. This statement is executed as part of the same global transaction. The COMMIT statement causes the transaction manager to commit the enitre global transaction via the two-phase commit process.

### 3.1.5 Summary

To summarize, the distributed access prototype supports single-level transactions in a multilevel environment. Since the transactions are single level, there are no problems with covert channels in the protocol itself. The security approach taken in this prototype is the most conservative possible, namely, all trust is centralized in the resource manager (server) and the underlying secure distributed computing infrastructure. The RM implements a policy that allows a subject to operate on a local portion of a distributed transaction only if the transaction was initiated by the same user-id as the current subject, and if the transaction has the same level as the current subject. The prototype polyinstantiates transaction identifiers to eliminate the possibility of these identifiers being used as a covert channel. The underlying MVTO algorithm prevents transactions at different levels from conflicting and therefore eliminates transaction conflicts as a possible covert channel. Finally, the architecture is based on open systems standards which leaves open the possibility of using a COTS transaction manager to integrate multilevel and single level resource managers.

## 3.2 Distributed Processing Prototype

The distributed processing prototype, shown in Figure 10, extends Trusted RUBIX to allow clients of a local server to transparently access data located on one or more remote servers. To support this capability, the Trusted RUBIX server is modified to include a distributed query processing capability. The security approach taken for providing this capability is the most conservative possible—the distributed query processor (DQP) is simply an untrusted application that runs above the Trusted RUBIX engine. The DQP supports an extended version of the SQL language that permits users to qualify table names with the resources (databases) in which they exist. This gives users the capability to formulate queries that span multiple distributed databases. When the DQP encounters a table with a resource qualifier, it will transparently connect to the remote resource to access the necessary data.

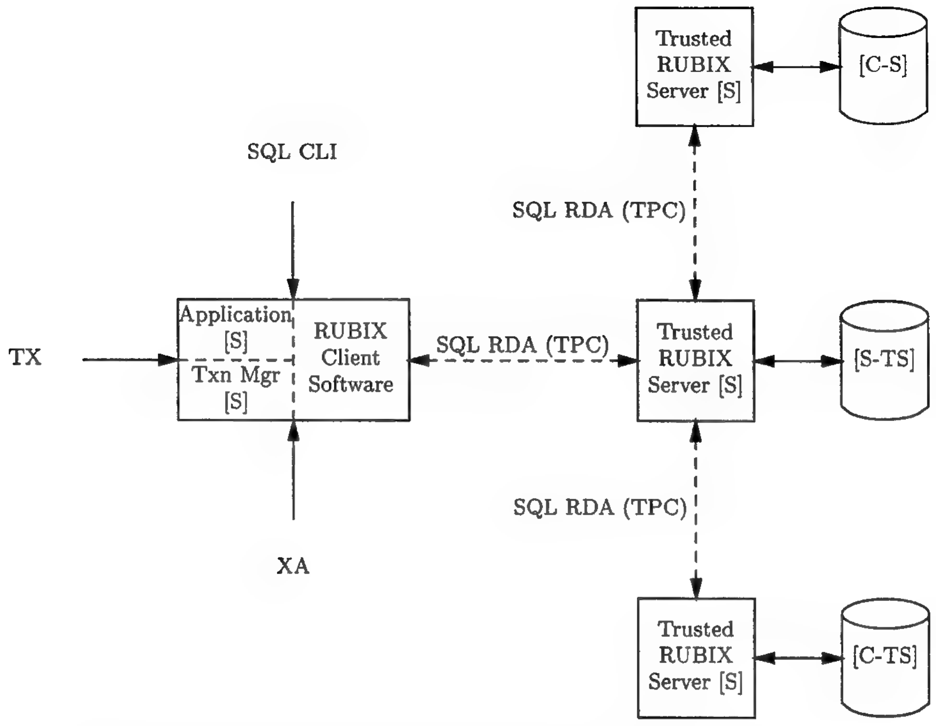

Figure 10: Distributed Processing Prototype Architecture.

To support the goal of providing location transparency, the SQL language is extended with data definition language (DDL) constructs that allow users to define integrated views of distributed data (federated schemata). Users can then submit queries on such federated schemata as if they were local objects. If the location of the underlying objects changes, only the federated schema needs to be updated—the applications that reference the federated schema are isolated from the change. The extensions that were made to the SQL language to support this prototype are described in [9].

A key feature of this architecture is that the DQP accesses remote servers in exactly the same way as do other clients. As discussed in section 2.1.2, Trusted RUBIX relies on the underlying secure distributed computing infrastructure for identification, authentication, and server invocation. The basic process is that the user is identified and authenticated on the client platform, and then the underlying secure distributed computing facilities are used to start up the untrusted portion of the Trusted RUBIX server on the server platform. This process is started up under the same user identifier and security level as the requesting client. In the DQP prototype, the untrusted portion of the server includes the DQP. Now, if there is a need for the DQP to access data located on another server, the new server is started up using exactly the same process. In this case, the DQP is actually acting as a client to the second server. The second server places no trust in the DQP and retains all responsibility for enforcing its security policy. This invocation process is shown in Figure 11. The remote server performs access checks based on the user's id and the level of the

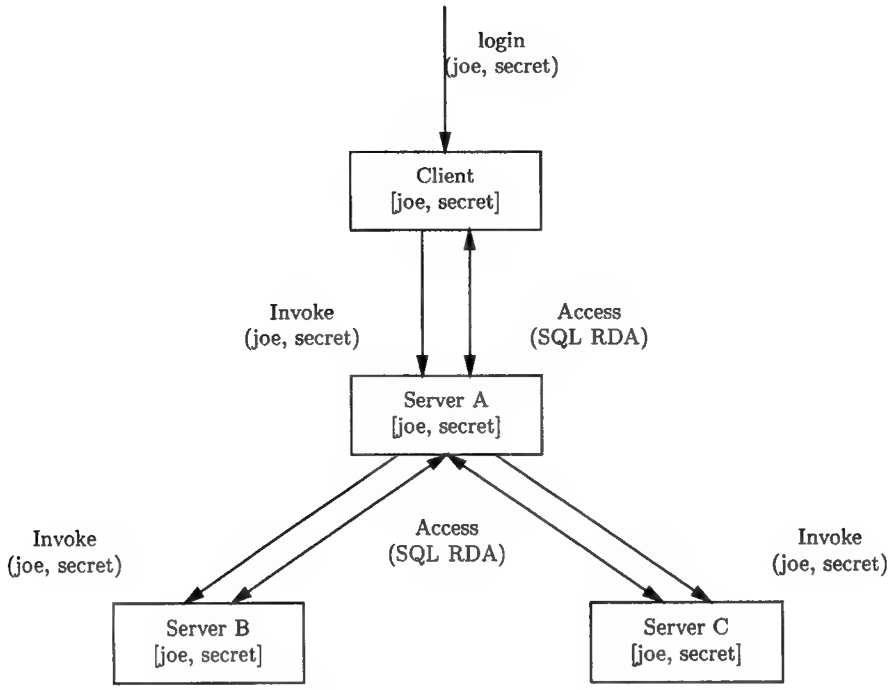

Figure 11: Recursive Invocation of Trusted RUBIX Servers

incoming connection. The DQP has no special privileges with respect to the remote server.

### 3.2.1 Distributed Query Processor Design Overview

**3.2.1.1 Schema Architecture** The DQP prototype is designed to minimize the amount of data that must be stored in a distributed database catalog. If no federated schemata are defined, then there is no need for a distributed database catalog. This is because the SQL standard "information_schema" [26] contains all the required descriptive information for each database. That is, the definition of an object stored in a particular database is available in the "information_schema" for that database. When the DQP needs descriptive information about objects in a particular remote database, it simply connects to that database and queries the local information schema.

There is a need for a distributed database catalog to describe federated schemata and their contained objects. Currently, federated schema consist of a set of imported relations, each of which can have an alias. The metadata describing these virtual objects is contained in a special schema called "dqp_schema". The structure of this schema is described in [8].

**3.2.1.2 Software Architecture** The internal design of a Trusted Distributed RUBIX server is shown in Figure 12. The architecture was developed by inserting an untrusted distributed query processing layer between the Trusted RUBIX connectivity software and the Trusted RUBIX SQL Engine. This distributed query processor consists of three functional

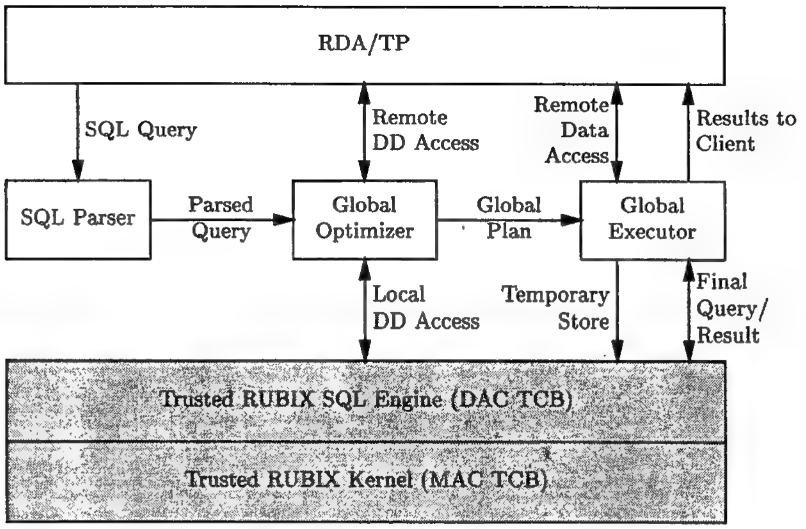

Figure 12: Trusted RUBIX Distributed Query Processing Architecture.

components:

1. *SQL Parser.* This parser parses queries expressed in the extended version of the SQL language.

2. *Global Optimizer.* This component creates an optimized execution plan for satisfying a query on distributed data. The plan generation process consists of 3 steps:

   (a) *Schema Transformation.* In this step, the optimizer translates a query over federated schemata into a query over objects in the external schemata in the referenced databases. To perform this step, the optimizer must access the description of the federated schemata.

   (b) *Semantic Validation.* In this step, the optimizer semantically validates the parse tree and resolves all partially qualified object references. To perform this step, the optimizer must access the descriptive information for the involved databases. This information is contained in the information_schema for the database. If the database is local, this information is obtained by directly accessing the SQL Engine. If the database is remote, this information must be obtained through RDA.

   (c) *Plan Generation.* In this step, the optimizer generates a plan to satisfy the query. The resulting plan has two parts: (1) a collection plan that describes what remote data has to be brought to the local site to satisfy the query (this data is stored in temporary tables), and (2) a final query over local data and remote data (now stored in temporary tables) that is equivalent to the original query. This constructs the final result for the user. The job of the optimizer is to generate a plan that minimizes the amount of information that is transferred across the network. The minimization can be done by transferring only the required tuples and attributes of the target relations. The required attributes are determined by the implicit or explicit references to the attributes in the query. The required tuples depend on the implicit or explicit predicates in the query. Simple predicates (e.g., $i > 10$) can be directly applied to reduce result sizes. The selectivity of join predicates can also be exploited using a technique know as semi-join reduction. This latter technique was not used in this version of the prototype.

3. *Global Executor.* The global executor takes the global plan, executes it, and returns results/status to the client. There are two phases to the execution process: data collection and final result construction. In the data collection phase, the Global Executor executes the subqueries in the data collection plan portion of the global plan. To execute a subquery, this component connects to the appropriate remote host, executes the query, and stores the result in a local temporary relation. In the second phase, the executor submits the final query to the local SQL Engine for execution, and returns results/status to the client.

### 3.2.2 Modifications To The Trusted RUBIX TCB

Two modifications were made to the Trusted RUBIX TCB to support the distributed query processing capability. These changes were not strictly necessary, but they made the resulting system much more usable. Both of these changes addressed problems related to temporary tables.

The first change addresses the problem discussed in section 2.3.2 where storing temporary results in tables can result in overclassification of final results due to the need to label the data at the level of the DQP process. The modification was to modify temporary tables so that a process can specify the label for a tuple inserted in a temporary table. This obviously solves the problem of label float because the DQP can now set the label in the temporary table to the label returned from the remote site. That this does not cause a security problem requires some explanation. This approach takes advantage of some special properties of temporary tables. Temporary tables in Trusted RUBIX are local to a session, that is, no other session can see them, and they are destroyed when the session is complete. This means that the temporary table can only contain data that the current process has access to since only that process could have put it there. As a result, the labels on a temporary table are informational only, and therefore, it is safe to allow the user to specify these labels.

The second change addressed the problem that if a user's current schema is not at the same level as the current session level, then distributed query processing will not work because a temporary table can not be created. The reason why a table cannot be created in a schema at a lower level is because of the problems it causes when the schema is dropped (e.g., DROP CASCADE). This problem does not occur with temporary tables because the are not explicitly dropped, they just go away at the end of a session. As a result the restriction can be removed for temporary tables and thus the temporary table creation problem is removed.

### 3.2.3 Example

The following examples shows the processing of a distributed query. The original query submitted by the user is as follows:

```
SELECT weapon_type, weapon_amount, weapon_description
FROM   wing_weapons@bw509 as ww, weapons@command as w
WHERE  ww.weapon_type = w.weapon_type
AND    weapon_type LIKE "AGM%"
```

Note that the tables referenced in the FROM clause are actually stored at different sites and are referenced in the query through resource qualifiers (table@resource). This query is decomposed by the DQP into the following subqueries:

```
temp1 <- SELECT * FROM wing_weapons WHERE weapon_type LIKE "AGM%"
temp2 <- SELECT * FROM weapons WHERE weapon_type LIKE "AGM%"
```

Note that the most restrictive possible WHERE clause has been included on each of these subqueries. This is to minimize the amount of data that needs to be transmitted across the network. The results of these subqueries are stored in a temporary table at the local site. The DQP transforms the original query to produce the following query over the temporary tables:

```
SELECT weapon_type, weapon_amount, weapon_description
FROM   temp1 as ww, temp2 as w
```

This query is executed on the local DBMS and the final result is returned to the user. The problem with the above example is that the user has to know exactly where the data is located. If the query is embedded in an application program, and the tables are subsequently moved, then the application program needs to be modified. Another approach is to create a federated schema that hides the location of the data. The following commands create a federated schema with two tables:

```
CREATE VIRTUAL SCHEMA inventory;
IMPORT TABLE weapon@command INTO inventory;
IMPORT TABLE wing_weapons@bw509 INTO inventory;
```

This schema is a virtual schema in that the data in these tables are not actually stored at the local site. These commands simply place information into a set of tables in the DQP data dictionary. The important thing to note is that a user can now use this schema as if the data were actually stored at the local site. The following reformulation of the above query is independent of the actual location of the data:

```
SET SCHEMA inventory;

SELECT weapon_type, weapon_amount, weapon_desc
FROM   wing_weapons, weapons as w
WHERE  ww.weapon_type == w.weapon_type
AND    ww.weapon_type LIKE 'AGM%';
```

If the data need to be moved (perhaps because usage patterns change), then only the schema needs to be changed. All the applications stay the same. Internally, the DQP takes a query on a federated schema and, using the federated data dictionary, transforms it into a query with resource qualifiers as shown below:

```
SELECT weapon_type, weapon_amount, weapon_description
FROM   wing_weapons@bw509 as ww, weapons@command as w
WHERE  ww.weapon_type = w.weapon_type
AND    weapon_type LIKE "AGM%"
```

This query is then processed as discussed above.

### 3.2.4 Summary

In summary, the distributed processing prototype provides integrated access to distributed multilevel data. This is provided by placing a distributed query processing capability on the Trusted Distributed RUBIX server. The prototype also gives users the capability to define federated schemata that provide integrated views of distributed data and isolate applications from the distribution of those data. The DQP is an untrusted application that runs on top of the Trusted RUBIX TCB. The DQP enforces no policy—all policy decisions are made by the local DBMSs. When a DQP access a remote site for a user, it accesses it using the users identity. The DQP accesses the remote site using the same invocation mechanism as is used by clients. Finally, temporary tables with informational labels are used in the prototype to avoid the problem of result overclassification.

## 4 Policy

Because of its architecture, Trusted Distributed RUBIX is *not* a trusted distributed system. It is an *untrusted application* that accesses multiple *independent* trusted DBMS servers. The composition of the trusted DBMS servers and the underlying secure distributed computing infrastructure form a trusted distributed system upon which Trusted Distributed RUBIX is hosted. Policy statements and formal models are not required for applications that are untrusted with respect to their underlying platform and that do not implement their own policies. Therefore, the Trusted Distributed RUBIX application requires no statement of security policy or formal model other than that which already exists for the platform on which it is placed.

However, since the Trusted RUBIX servers are trusted with respect to the underlying infrastructure, they represent a modification to an existing trusted distributed system, and as such, potentially change the trust characteristics of that system. As a result, before fielding such a system, an analysis must be conducted to determine the properties of the composite system. An important input to this analysis is a formal statement of the individual component policies. Section 4.1 discusses the Trusted RUBIX formal model[27]. Section 4.2 discusses the issues of composing this policy with that of the underlying infrastructure.

## 4.1 Security Policy Model

The Trusted RUBIX model is an adaptation of the SeaView security model which was developed at SRI International [28]. The SeaView model was selected as a basis for the Trusted RUBIX model for several reasons. First, the SeaView model is the most complete DBMS access control policy model available to date. Second, this model has achieved a substantial degree of acceptability in the secure DBMS community, and, therefore, cannot become an issue of contention or concern during a security evaluation. Third, this model can be used in a wide variety of DBMS system interpretations because of its generality and abstraction. Lastly, the model has benefited from the critical scrutiny of the research community for quite some time now and, consequently, it has become sufficiently stable for widespread use.

The most significant modification to the SeaView model that has been made for Trusted RUBIX is the different choice of model-component composition. This modification reflects the different relationship between Trusted RUBIX and the underlying UNIX Trusted Computing Base (TCB) model. The SeaView model was originally developed for the SeaView prototype, which runs as an untrusted application above an evaluated TCB [29]. As such, the model reflects a system where the underlying TCB enforces a mandatory security policy, and the DBMS enforces higher-level policies (e.g., discretionary access control). In the SeaView model, the first model layer (the MAC Model) corresponds to the underlying reference monitor, and the second model layer (the TCB Model) corresponds to the DBMS. The Trusted RUBIX architecture differs from the SeaView architecture in that a portion of the DBMS runs as a trusted subject with respect to the underlying TCB as described above. In the Trusted RUBIX model, the MAC Model corresponds to the composition of the underlying reference monitor and the Trusted RUBIX Kernel, and the TCB Model corresponds to the SQL Engine. The result of this difference is that a number of the types and properties that were in the TCB Model in the SeaView version of the formal model have been moved to the MAC Model in the Trusted RUBIX version.

The other modifications to the model reflect different policy choices that were made in the development of the Trusted RUBIX TCB. The following are the major differences between the SeaView and Trusted RUBIX access control policy:

- Differences in labeling and integrity rules relating to the fact that Trusted RUBIX labeling is at the tuple level.
- The introduction of a schema object and associated set of privileges.

- The association of a set of privileges with the database object.
- Different group semantics whereby a subject can be simultaneously in multiple groups.
- The introduction of a special public group which has as its membership all users.
- The select and insert privileges are permitted at the table and column level.
- The addition of the references and admin privileges.
- The elimination of the mandatory integrity component of the policy.
- The elimination of application specific constraints from the policy.

These differences result in significant modifications to the detail of the model, however, the basic structure of the model is preserved.

## 4.2 Security Policy Composition

The goal for security policy composition is a precise and systematic characterization of a security policy which is composed of several independently specified policies. In Trusted Distributed RUBIX, security level composition must be done on multiple levels:

- The Trusted RUBIX SQL Server must be composed with the Trusted RUBIX kernel.
- The Trusted RUBIX kernel must be composed with the underlying operating system TCB.
- The underlying operating system TCB must be composed with the other TCBs in the secure distributed computing infrastructure.

To make progress toward the goal of a precise overall characterization of the access-control policy for Trusted Distributed RUBIX, we have developed a framework that allows security policies to be composed in an informal, yet precise and systematic, manner. This framework consists of two parts. The first part is a framework for security policy definition that is general enough to characterize a wide variety of policies. The second part is a set of policy composition rules which are based on the policy areas and rules specified in the policy definition framework. To date, composition rules for different policies, or for different components of the same policy, have not been proposed or investigated. Although we do not propose any specific set of composition rules, we do illustrate the need for such rules and some of the pitfalls attendant to defining such rules. The details of this framework are presented in appendix A.

## 5 Improvements To Trusted RUBIX

As part of this effort, a significant amount of analysis and engineering work was conducted on the Trusted RUBIX server. Two of the most important results of this work were the conflict-secure MVTO mechanism described in section 2 and the temporary table capability described in section 3. This section describes the other significant results of this effort.

### 5.1 Security Engineering

A important part of the Trusted Distributed RUBIX effort was continued security analysis and engineering of the Trusted RUBIX server. This work was a continuance of the work documented in [30, 31]. The result of that work was a restructuring of Trusted RUBIX into a multi-subset architecture as described in Section 2. The work conducted under this effort was to re-engineer both the Trusted RUBIX kernel and the SQL engine to improve its modularity. The result is a system that consists of a set of well-defined and largely independent modules. The analysis of the Trusted RUBIX system also uncovered some general areas for improvement that could not be addressed during this effort.

The first general area is Trusted RUBIX Kernel minimization. There are currently functions in the Trusted RUBIX kernel that can just as easily be supported in the SQL Engine and thus further minimizing size of the kernel. Three major items were identified. The first was typed records. The current interface to the Trusted RUBIX kernel is an interface based on typed records. This implies that all the software that deals with relation metadata, field types, type conversions, and conversions to storage formats is in the Trusted RUBIX kernel. This software could be removed from the Trusted RUBIX kernel by modifying the interface so that it supports untyped records. The result would be an interface where a record consists of a binary stream of data of which a fixed portion is recognized as key.

A second set of functions that can be removed from the RUBIX kernel are those dealing with schemata. In the Trusted RUBIX kernel, schema are used as a way to impose a hierarchical structure on relations (similar to a UNIX directory structure). This hierarchical structure could just as easily be supported in the SQL engine through a set of system-maintained relations. The only disadvantage is that, with the change, the RUBIX kernel would only support a flat name space. This is actually not a problem because the entire concept of relation names can also be removed from the RUBIX kernel. This is discussed below.

The final component identified for removal from the Trusted RUBIX kernel is all functionality associated with user supplied names. The result is a system where, when an object is created, the system provides the user with a name for that object. Having the system generated name for an object means that the system doesn't have to cope with user-supplied name collisions across security levels (e.g., by polyinstantiating relation names). The mapping of user-supplied names to system generated ones can be implemented in the SQL Engine using relations to store the mapping. For example, named schemata, and named relations could be supported in the SQL Engine with the following two tables:

```
SCHEMATA(SCHEMA-NAME, ...)
TABLES(SCHEMA-NAME, TABLE-NAME, sys-table-name, ...)
```

The SCHEMATA table is used to store all the user defined schemata and the TABLES table maintains the relationship between tables and schemata, and user-supplied table names and system-generated table names [1]. The choice of polyinstantiating names or not can be done by creating the TABLE relation with the appropriate polyinstantiation discipline[2, 3]. For example, if the relation is created with the discipline POLYNONE, then attempting to create a table in a schema that already has a table by that name will cause a key conflict and thus the table name will not be polyinstantiated.

The second general area for improvement was layering. The re-engineering conducted during this effort resulted in a more modular architecture. However, there are some cases where there are cycles in the module dependency structure. Perhaps the most serious case of this problem is in the naming of files by the file manager. File names are currently implemented using the relation manager which in turn uses the file manager to store relations. Removal of these cycles from the dependency structure will result in a system that is easier to modify, analyze, and test.

The incorporation of the above changes into the Trusted RUBIX kernel will result in the architecture shown in figure 13. The major difference between this and the current Trusted RUBIX kernel architecture are the changes discussed above.

## 5.2 Performance Engineering

Another important part of the Trusted Distributed RUBIX effort was performance analysis and engineering of the Trusted RUBIX server. There were two major results of this effort. The first was an analysis of the current buffer management scheme and a proposal for and improved scheme. The second was a set of improvements in the was that Trusted RUBIX performs query optimization. These changes are discusses in the following sections.

### 5.2.1 Buffer Management

**5.2.1.1 Analysis of Current Implementation** The current situation is that each instance of Trusted RUBIX maintains a buffer pool for each open table. This situation is shown in figure 14. When a read/write is requested, the buffer pool is checked to see if that page is already "in memory". If the page is in the buffer pool, the read or write is done directly to that page. If the page is not in the buffer, a check is made to see if there is free space in the buffer pool. If there is free space, the requested page is read into the pool. If there is no space in the buffer pool, an old page is forced to disk and removed based on a least recently used algorithm. The requested page is then read into the vacated slot. When the table is closed, all the buffers are forced to disk and deallocated.

Since each process maintains its own buffer pool, there is the problem of maintaining a consistent view of the database in the face of read/write activity of concurrent processes.

[1]In the above definitions, key attributes are in upper case.

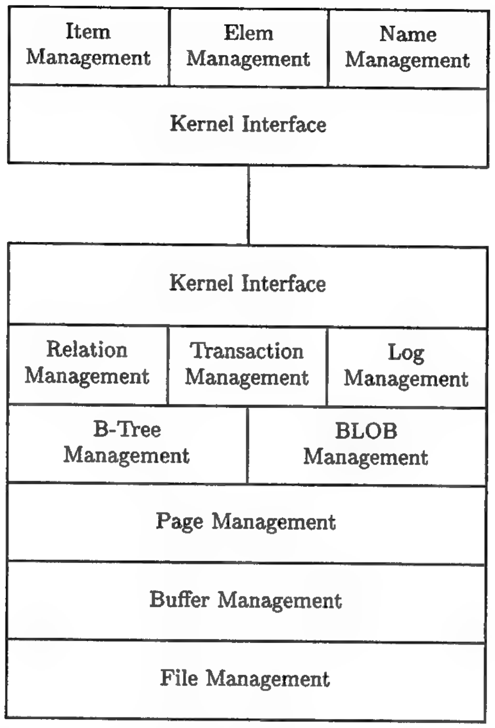

Figure 13: Trusted RUBIX Kernel Architecture.

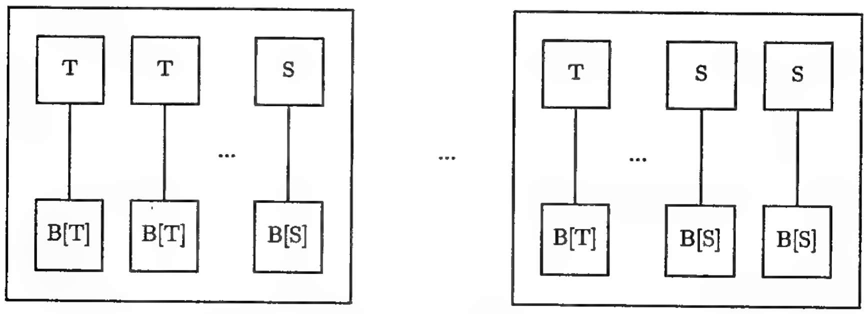

Figure 14: Current Buffer Implementation

This problem arises because, if writes are done only to a process's local buffer, then those writes will not be visible to other processes. This problem is solved using a pair of functions called btgrab() and btletgo(). Before a process does a read it calls btgrab() which places a short term lock on the B-Tree, looks at the B-tree header, and if the file has changed, marks all its buffers as invalid. This lock is released once the read is complete (btletgo()); Similarly, before a process does a write, it grabs the B-tree. It then does its write to the copy of the page in its buffer. When it has finished the write, the process calls btletgo() to release the B-tree. Btletgo() flushes all modified pages to the disk so subsequent readers will see them. The lock done by the btgrab() ensures that other processes do not attempt to read the B-tree while a page is being written (or the tree is being reorganized).

Another interesting aspect of the current buffer implementation is that buffer management is done only for B-trees. The result is that objects that are not stored as B-trees are not buffered (e.g., Free-space lists and BLOBs).

There are a number of inefficiencies in the above scheme:

1. Multiple accesses to the same table within an instance of Trusted RUBIX are buffered independently.
2. Accesses to tables in different instances of Trusted RUBIX are buffered independently.
3. A modification to the B-tree currently invalidates all pages in a buffer pool. All that is actually necessary is to invalidate the modified pages.
4. Checking the lock on the B-tree and reading the B-tree header costs an extra I/O on each access (even if no pages need to be re-read).
5. The entire B-tree is locked even in cases where page level locking would suffice.
6. Certain types of file access are not buffered at all.

These items adversely impact performance because they result in a larger number of I/Os than necessary.

**5.2.1.2 Improved Buffer Management Scheme** This section proposes a new buffer management scheme that addresses the problems noted in the previous section. The first characteristic of the new buffer management scheme is that the buffer management capability is extracted from the B-tree implementation and is layered directly on top of the file manager. This situation is depicted in figure 15. As a result of this placement, *all* I/Os will be buffered under the new scheme, not just B-Tree I/Os. The second characteristic of the new scheme is that there is a single buffer pool that is shared by all instances of Trusted RUBIX. This buffer pool in maintained in shared memory as shown in figure 16. There are two advantages to a shared buffer pool. The first is that the efficiency (hit ratio) of buffering will be improved. That is because a process that accesses a page that was previously accessed by another process (or the same process on a previous open of the relation) may

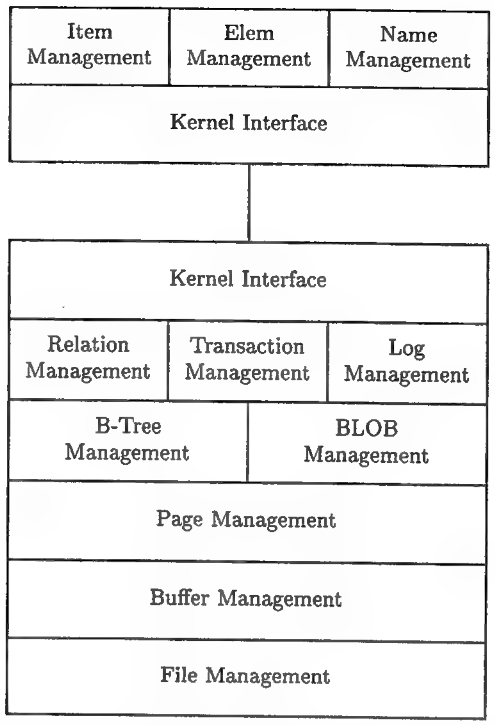

Figure 15: Layering of Trusted RUBIX Kernel

find the page already in memory. The second is that buffer pages do not need to be flushed and re-read in order to keep multiple independent buffer pools in synchronization. This will result in a significant reduction in the number of I/Os. A third characteristic of the new scheme is that locking is performed at the page level rather than at the file level. This will result in a significant increase in the amount of concurrency supported by the system. The B-tree implementation can be revised to utilize existing protocols for page-level locking of B-trees [32, 33].

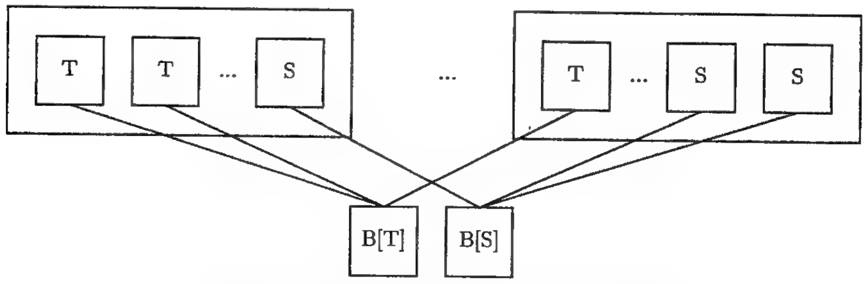

Figure 16: Shared Buffer Pool Scheme

### 5.2.2 Query Optimization

The way in which Trusted RUBIX handled query execution had a number of deficiencies. The most notable deficiency was the fact that secondary indexes were not utilized on selections and certain joins. The result was that Trusted RUBIX spent a large amount of time unnecessarily scanning entire relations.

Consider the following selection query:

```
SELECT * FROM R
WHERE R.a = 5;
```

If R.a is the primary key of the relation R, at most one tuple can satisfy the predicate, which can be retrieved by a seek into the primary index. Even if R.a is not the primary key of R, the number of tuples satisfying this predicate should be a fraction of the total number of tuples present in R. If there exists an index on R.a for R, a seek through the index should retrieve only the matching set of tuples, avoiding a total scan over R. For other comparison operators also, the number of tuples retrieved when an index is utilized should be much smaller than the total number of tuples. Therefore, utilizing an index can significantly reduce query execution time if index support is provided.

Accessing a relation with index support is a two step process. The first step is to select an index from the existing set of indices (including the primary index). This selection is based on the attribute(s) in the selection predicate. The second step is to determine a range of values to be retrieved. These values will be determined by the operators and constants in the selection predicate. Only tuples having values within that range will be retrieved. Tuples having values outside that range will certainly not satisfy the predicate, and hence will not be retrieved at all. This will reduce the number of unnecessary tuples retrieved during a selection, and, therefore, will reduce execution time. We call the restricted retrieval a *restricted select.*

Support for theta join optimization uses this restricted select function. A theta join combines a cross product with a select. Consider the following query:

```
SELECT *
FROM  R,S
WHERE R.r = S.s;
```

If the query is implemented as $\sigma_{R.r=S.s}(R \times S)$, then the selection predicate "$R.r = S.s$" is tested on a relation of size $|R| * |S|$, where $|R|$ and $|S|$ are the cardinality of R and S, respectively. The optimized theta join "unwinds" the cross product and uses a select for each element in the first table. This unwinding is based on the following equivalence:

$$R \bowtie_\theta S \equiv \bigcup_{i=1}^{|R|} \left[ \{R[i]\} \times \sigma_{\theta^{R.x}_{R[i].x}}(S) \right]$$

where R[i] is the ith tuple of R and $\theta^{R.x}_{R[i].x}$ is $\theta$ with all occurrences of R.x replaced with R[i].x. Appling this equivalence to the query in our example yields:

$$R \bowtie_{R.r=S.s} S \equiv \bigcup_{i=1}^{|R|} \left[ \{R[i]\} \times \sigma_{R[i].r=S.s}(S) \right]$$

The selection on $S$ can now be performed using the previously implemented *restricted select*, using the primary or a secondary index. This can result in significantly faster execution times depending on the selectivity of $\theta$.

Another deficiency in the previous query execution scheme was that it executed the query tree generated by the parser as is, without any modification. The parser generates a canonical query tree corresponding to the stated SQL-query. Such a canonical tree can be very inefficient to execute because, for example, it produces a cartesian product for any implicit join, and applies all selection predicates after that cartesian product. Query execution was optimized by reorganizing the parse tree by applying a set of heuristics. Reorganization is possible because of the commutativity of the relational algebra operations under certain conditions.

Cost-based optimization (e.g., assigning a cost to a set of equivalent query plans and selecting the least cost plan) was not addressed as a part of this effort. This type of optimization requires database statistics and cost-estimation functions. Trusted RUBIX does not currently keep database statistics and we do not have the required cost estimation functions. Implementing statistics maintenance and developing accurate cost-estimation functions are non-trivial tasks that were not within the scope of this effort. Note that, heuristic based optimization is performed before cost based optimization in a two stage optimization process. Therefore, if we choose to implement cost-based optimization in the future, the current work will not have to be discarded.

## 5.3 Transaction Savepoints

A transaction savepoint is a point in a transaction history to which the transaction can be rolled back. Normally, transactions can either be rolled back or committed. A transaction savepoint mechanism allows transactions to be partially rolled back. This is an important feature when the transaction has done a significant amount of work, yet a portion of it must be rolled back. The primary motivation for developing the savepoint mechanism was for internal use in the implementation of the Trusted RUBIX SQL engine. The savepoint mechanism is an elegant way for the system to clean up the database state after an SQL statement fails (e.g., as the result of a violated integrity constraint). Before an SQL operation is started, the system creates a savepoint. If the operation fails for any reason, the system can rollback to that savepoint and the state of the database will be as if the operation was never submitted. A benefit of this approach is that the SQL Engine does not need a lot of special purpose code to cleanup after different types of operations, and hence the system is smaller, less complex, and more robust. The savepoint mechanism was designed in such a way as to not introduce covert channels into the secure MVTO algorithm described in section 2.

## 6 Conclusions

This document presented the results of the Trusted Distributed RUBIX project. This project was a research effort to address the practical issues of developing a high-assurance multilevel secure distributed database management system. The approach followed in this effort was to investigate these issues by developing a prototype multilevel secure distributed DBMS. The prototype was developed by extending Trusted RUBIX to support two forms of distributed database access. These capabilities were provided in two separate prototype increments: the distributed access prototype and the distributed processing prototype. The distributed access prototype provides a single-level atomic multi-site transaction processing capability. The distributed processing prototype provides users with integrated access to distributed multilevel data. Both of these prototypes have high-assurance architectures based on open systems standards.

The lessons from the distributed access prototype are:

- It is possible to provide a multilevel secure transaction processing capability without trust in the Transaction Manager.
- Extending a Resource Manager to support distributed transaction processing can introduce new covert channels.
- The global transaction identifier covert channel can be eliminated by polyinstantiating transaction identifiers.
- It is possible to develop an architecture for distributed transaction processing in a multilevel environment that is compatible with open systems standards.

The primary lessons learned from the distributed processing prototype are:

- It is possible to develop a distributed query processor that is completely untrusted with respect to the underlying DBMSs.
- Multilevel distributed query processing can lead to result overlabeling.

Finally, some general conclusions:

- High-assurance distributed data management is feasible in the near-term.
- Secure distributed computing infrastructures need improvement.

## 7 Future Work

There are a number of directions in which this work can be extended. First, there are a number of areas where additional work can be done in secure distributed data management. These include:

- *Replication.* Most commercial DBMSs have a data replication capability in their non-secure products. There are a number of issues that must be addressed before supporting such capabilities in a high-assurance MLS D-DBMS. The issues to be investigated are both architectural and policy related.

- *Heterogeneity.* It would be desirable to use an MLS D-DBMS to integrate data from multiple heterogeneous data sources. However, before this can be done, there are a number of issues that must be addressed. These include indirect authentication, heterogeneity in security attributes, and heterogeneity in MLS data models.

A second area where additional work could be done is improving the prototype developed as part of this effort. Areas for future work include:

- *Distributed Query Optimization.* The optimizer implemented in this prototype does relatively simple heuristic optimizations. This optimizer could be improved by extending it to perform cost-based optimizations that take advantage of the selectivity of joins (semi-join reduction), and the parallelism inherent in a distributed DBMS.

- *Federated Schema Definition Language.* The federated schema definition capability implemented in this prototype is a relatively simple one where a federated schema is simply a set of references to tables stored at other sites. The only transformations permitted on the data are transformations on the table names. A more sophisticated capability could be developed where much more complex transformations could be supported. Such a capability would help the user deal with semantic heterogeneity.

Finally, there are a number of areas where work can be done to enhance the Trusted RUBIX server which is the base of Distributed RUBIX. These include:

- *Security Engineering.* There are a number of improvements that can be made to Trusted RUBIX in the area of modularity, layering, and TCB minimization that would improve its assurance characteristics. These improvements are discussed in detail in section 5.1.

- *Performance Engineering.* There are also a number of improvements that could be made to Trusted RUBIX to improve its performance characteristics. These include improved buffer management, cost-based query optimization, and general performance tuning. These improvements are discussed in detail in section 5.2.

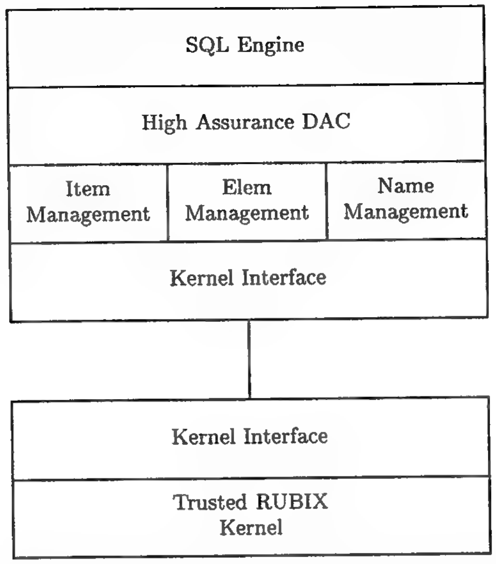

Figure 17: Architecture For High Assurance DAC.

- *High Assurance DAC.* In [34], a high-assurance discretionary access control mechanism called P-Views is proposed. Adding this mechanism to Trusted RUBIX would provide a high-assurance DAC mechanism to complement the current high assurance MAC mechanism. This mechanism could be implemented below the existing SQL DAC mechanism so users could continue to use the standard DAC mechanism. The SQL DAC would be constrained by the assured DAC policy. The resulting architecture in shown in figure 17.

- *Content-Based MAC.* The same properties that make P-Views a good mechanism for assured DAC (safety and simple mechanism), make them a strong candidate for a mechanism for content-based mandatory access control. The strict separation of MAC and DAC TCBs would make it relatively easy to prototype such a mechanism in Trusted RUBIX.

# Appendix A

# A Composing Security Policies

## A.1 Introduction

### A.1.1 Purpose

The principal purpose of this report is to define a framework for security policy composition for both centralized and distributed systems in an informal, yet precise and systematic manner. The notion of the security policy used throughout this report is that of the "technical security policy," as opposed to that of the "organization security policy" [35]. The immediate, practical motivation for security policy composition is the precise and systematic characterization of centralized and distributed Trusted RUBIX security policy, which is composed of several independently specified policies [30, 27, 1, 36]. To accomplish this goal, however, several subgoals have to be reached. First, a precise, systematic approach to policy composition requires the precise, systematic definition of a technical security policy. This subgoal has not been achieved by computer security research and development to date. Instead, only selected aspects of a security policy have been emphasized by various policy models and policy requirement specifications (e.g., in [12, 37, 11]. Hence, a security policy framework must be defined that is general enough to characterize a wide variety of policies.

Second, a set of policy composition rules must be defined. If all the component policies, and if all the components of an overall policy, are defined using the policy framework, it becomes necessary to base the composition rules on the policy areas and rules specified by the framework. To date, composition rules for different policies or for different components of the same policy have not been proposed or investigated. Although we do not propose any specific set of composition rules, we do illustrate the need for such rules and some of the pitfalls attendant to defining such rules.

Third, the proposed policy-definition framework and composition rules must be general enough to enable a designer to make credible statements of the overall policy of a trusted system once the individual policies or policy components have been shown to support the overall policy. While this goal was first articulated in the context of policy partitioning and composition proposed by the NCSC [37], a precise, systematic process for achieving this subgoal has not been proposed to date.

Without achieving the three subgoals mentioned above, the immediate, practical goal of defining the overall access-control policy for both centralized and distributed Trusted Rubix systems in a precise and systematic manner would continue to remain unfulfilled. The work reported in this report focuses on the definition of a general security policy framework and on articulating the need for composition rules within this framework. While the security policy definition within the proposed framework is informal, the framework provides a useful guideline for uniform and precise security policy definition. We illustrate the use of this framework with examples of security policies and composition of security policies.

### A.1.2 Outline

This report is organized as follows. In section 2, we motivate the need for providing a general framework for characterizing access control policies, review the differences between the policy composition problem addressed in the literature to date and our work, and delimit the scope of the problem addressed in this report. In section 3, we present a policy definition framework, and we illustrate its use with examples of common policy outlines. In section 4, we conclude this report by presenting a set of reasons as to why the proposed framework is complete. Section 5 contains the references cited in text. The appendix of this report contains examples of access control policy composition and their pitfalls.

## A.2 Motivation

The need for defining a specification framework for access control policies arises for the following reasons. First, a systematic way of specifying access control policies is necessary for the analysis of these policies. Analysis is necessary to avoid inadequate, incorrect or incomplete specifications. Second, a common framework for specifying access control policies is necessary for comparing access control policies and for determining their relative merits independent of the stated policy aims. Third, the specification framework is needed for the analysis of access control policy changes and the effect of those changes on the overall system security. Fourth, a common policy framework is necessary for composing access control policies. More specifically, without using a common policy framework it is difficult to provide a precise statement of the overall, composed access control policy and even more difficult to argue that the resulting policy satisfies stated policy goals. Moreover, without such a framework, the composition rules could not be defined in any systematic manner.

The composition of access control policies requires the composition of policy area rules in a systematic, precise manner. The interaction between different policy rules within the same policy area and between different areas can lead to inadequate composed policies. These policies can violate the protection warranted by the individual policies that are composed. We illustrate such inadequate compositions in the Appendix to this report.

### A.2.1 Differences between this composition problem and previous work

Previous work on formal policy composition has focussed primarily on the problem of composing components with a specific property in the process of building larger systems that preserve that property. For example, if one takes the outputs of a multilevel secure system and feeds them into the inputs of another multilevel secure system, and if the security levels of those outputs are the same as those of the inputs, one would want to conclude that the resulting system is multilevel secure. A very significant amount of attention has been devoted to this problem, called the "hookup security problem," since McCullough introduced it in 1987 [38]. McCullough showed that noninterference models, which essentially solved the problem of identifying covert channels in multilevel systems [39, 40, 41], cannot be applied to nondeterministic systems [42]. He also introduced a hookup security property for multilevel systems, called "restrictiveness," which applies to nondeterministic systems and is generally composable. That is, sense that if two systems are restrictive, so is their composite under the same-level input-output composition rule mentioned above. Johnson and Thyer slightly relaxed the "restrictiveness" property by introducing the "forward correctability" property for multilevel systems, which is also preserved by the composition rule mentioned above [43]. The forward correctability is a slightly weaker property in the sense that it is satisfied by more systems [44]. A review of much of the composability work based on variations of the hookup property is done by Lee, Boulton, Thompson and Soper [45].

Our work differs from composability under various forms of the "hookup" property in two fundamental ways. First, we consider the problems of composing different security policies within a definable, overall trusted system policy. Second, we also consider the composition of different areas of a single security policy, each area being supported by a partition or a subset of a given policy. While this latter problem has been pointed out by the TNI and TDI [37, 11], precise, systematic methods for solving it have been unavailable to date.

A recent noteworthy problem of security policy composition has been addressed by Hinton and Lee [46]. Their work examines the problem of assessing whether different policies that must be composed are "compatible." Here compatibility is being defined under non-hierarchical composition (e.g., subset composition discussed below) whenever the validity of one policy property will not preclude that of another. For example, if two policy properties independently authorize a "write" action and their non-hierarchical composition does not, the two policy properties are incompatible.

Our work focuses on different aspects of the composability problem than that addressed by Hinton and Lee. We consider access control policies that are compatible by definition or can be configured in a compatible way. We illustrate in the Appendix to this report further composability problems beyond compatibility. For example, we show that the non-hierarchical composition of compatible policies may violate the properties preserved by individual policies. For example we show that if we compose a typical multilevel secrecy policy and a multilevel integrity policy, the resulting composite introduces covert channels that did not exist in either of the two policies.

We believe that every one of the research subgoals mentioned in the previous section represents an unsolved research and development problem in its own right. That is, solutions to the problems of providing (1) precise systematic methods for policy definition, (2) composition rules, and (3) methods for defining the overall composite policies of trusted systems have been elusive.

### A.2.2 Scope

In this report we address the first of the three problems mentioned above, namely that of providing a uniform, precise framework for security policy definition. This framework is informal and represents a set of guidelines for classifying and defining security policy rules. These rules further define the security policy properties. In addition, we provide composition examples that illustrate the need for precise composition methods and rules.

We do not propose nor recommend any general composition rules. This represents an area of future research. However, our policy composition examples use specific composition rules, such as authorization rule conjunction and precedence and attribute-administration rule union.

While a formal theory for policy specification and composition is beyond the scope of this work, we believe that the development of such a theory is a worthwhile goal. The theory would consist of a formal specification framework within which one would be required to provide formal specifications for rules in each of the areas required by the framework. Further, the theory would require that formal composition rules be provided for each policy area so that the composed policy rules of each area could be formally derived and analyzed. Within such a formal framework, the composed formal policy becomes precisely defined and the resulting policy rules could be formally analyzed to determine whether these rules satisfy the goal of the overall policy. The soundness and completeness of the policy rules may also become determinable.

A noteworthy paper that addresses some fundamental aspects of composing formal specifications is that of Abadi and Lamport [47]. We believe that their approach could be used as a basis for defining composition rules based on formal rule specification. However, significantly more work must be performed to determine restrictions for the composed rules based on security semantics. For example, it is unclear whether the Abadi and Lamport approach to specification composability could express the restrictions that would be necessary to eliminate the covert channels presented in the examples of the Appendix.

## A.3 A Characterization of Security Policies

The security policies of a centralized or distributed system include Identification and Authentication, Access Control, and Audit. Each policy can be further divided into policy areas that consist of policy rules. Whether expressed as formal constraints or informal requirements, these rules represent the properties of the security policy defined within the proposed framework.

### A.3.1 An Access Control Policy Framework

Access control policies can be characterized in terms of five policy areas, namely

- definition of subject and object policy attributes,
- authorization of subject access to objects,
- administration of the policy attributes,
- subject and object creation and destruction, and
- object encapsulation.

These policy areas, which are defined in the following subparagraphs, can be used to characterize a wide variety of security policies, including traditional discretionary and non-discretionary policies. The intent of characterizing all security policies in terms of these five areas is to provide a general framework for policy specification and composition applicable to all policies regardless of the aim of those policies.

Within a policy area, a policy requirement is stated as a rule or a set of rules. These rules define the properties of each policy area. If individual policies will follow the proposed framework, compositions of individual policies can also be defined in terms of the composition of the policy rules in the five areas. Hence, whenever multiple policies are supported within the same system, these five areas define the composition of policies and how the policies are enforced (e.g., subject and object type coverage, precedence of enforcement).

Access control policy areas may include rules that may not be applicable to some policies. In such cases, the individual rules shall be designated as non-applicable in the definition of the policy. For example, the transitive distribution of permissions applies primarily to discretionary policies. Consequently, attribute administration rules of non-discretionary controls may not include conditions for transitive distribution and revocation, and these conditions will be designated as non- applicable to a specific non-discretionary policy. Similarly, discretionary policies may not necessarily control access to object status variables (e.g., existence, size, creation, access and modification time, locking state). Hence, the rules or conditions specifying such controls may be designated as non-applicable in specific discretionary policies.

Some policy area may include rules that may not be applicable to some types of objects. In such cases, the individual rule that is applicable to that type of object will be specified separately. The intent of providing per-type access policy specifications is to capture the access control needs of a particular type of object without imposing impractical or meaningless policy constraints. For example, user-oriented rules for access-right administration need not be imposed on objects that cannot, and are not intended to, store user data. Requiring transitive, temporal, time- and location-dependent distribution and revocation conditions for a discretionary policy on interprocess communication objects such as semaphores and sockets or on publicly accessible objects such as bulletin boards would be both impractical and unnecessary. However, when per-type specifications are used, the totality of the per-type rules and conditions must be shown to support the policy properties.

**A.3.1.1 Definition of Policy Attributes.** A policy specification must define the subject and object attributes required by that policy, and must identify the context-dependent policy attributes. Subject attributes may include user-dependent credentials (e.g., user identifier, group or role identifier(s), confidentiality or integrity levels, access time intervals, access location identifier), as well as user-independent credentials (e.g., system privileges allowing the invocation of TCB functions unavailable to unprivileged subjects). Object attributes may include user-dependent, policy attributes (e.g., distinct object permissions for different users), as well as user-independent attributes (e.g., secrecy or integrity privileges accessible only to privileged processes). Finally, context-dependent policy attributes may include the current time, group definitions, and/or a level indicating whether an emergency is in progress.

If multiple policies are supported, the rules for defining subject and object attributes must partition these attributes on a policy basis.

**A.3.1.2 Authorization of Subject References to Objects.** A subject's reference to an object consists of invoking an action on a set of objects. The subject's reference to the object can be thought of as a request to access that object. Examples of actions include invocations of TCB commands, function calls, processor instructions, protected subsystems, and transactions. An action may have separate policy attributes from those of the issuer of the reference. For example, invocations of transactions and protected subsystems (which encapsulate objects) will generally include policy attributes that differ from those of their invokers. In contrast, other actions such as invocations of individual processor instructions, TCB function calls, some TCB commands, and applications programs are prohibited from using policy attributes, such as identity, group, role, or secrecy and integrity levels, that differ from those of their invoker. Policy attributes involved in rules for deciding access authorization are referred to as "access control" attributes.

The rules for authorizing subject references to objects can be defined in terms of (1) the subject's authorization to an action, (2) the action authorization to one or more objects, and (3) the subject's authorization to one or more objects. These rules are based on the policy attributes defined for subjects and objects. The rules can be defined either on <subject, action> and <action, object(s)> tuples or on <subject, action, object(s)> triples, depending upon the specified policy. The authorization rules specify the authorization scope and granularity in terms of (1) resources containing one or more objects, (2) individual subjects and objects, (3) the subject and object policy attributes, and (3) the subject and object status attributes (e.g., existence, size, creation, access and modification time, locked/unlocked). The authorization rules also specify whether delegated authorization (i.e., authorization of a subject access performed on behalf of other subjects, using combined-subject attributes) is allowed.

The coverage of the authorization rules is specified in terms of the types of objects and subjects to which they apply. If different rules apply to different subjects and objects, the totality of these rules is shown to support the defined policy properties. If multiple policies are supported, these rules define the composition of policies and how the authorization conditions are enforced (e.g., subject and object type coverage, order of enforcement)

**A.3.1.3 Administration of Policy Attributes.** A policy specification must include rules for administering and maintaining the subject and object attributes. Rules for administering policy attributes determine the conditions under which a subject can change its own attributes as well as those of other subjects and objects. These conditions define whether a subject is authorized to modify a policy attribute and may not rely on those used in the authorization of subject references to objects (discussed above). Otherwise, a cyclic dependency may arise between the attribute administration rules and those of authorization of subject references to objects. Attribute maintenance rules also define the attributes for subject or object import or export operations.

As an example of attribute administration rules, consider those rules that determine what subjects have the authority to distribute, revoke, and review policy attributes for specific subjects and objects, and the conditions under which these actions can be performed.

The distribution and revocation rules determine which of the following conditions are enforced.

1. Selectivity: distribution and revocation can be performed at the individual attribute level, such as user, group, role, permission, privilege, security or integrity level.

2. Transitivity: a recipient of a permission from an original distributor can further distribute that permission to another subject, but when the original distributor revokes that permission from the original recipient, then the subject which received that permission from the original recipient will also have it revoked.

3. Immediacy: the effect of the distribution and revocation of policy attributes should take place within a specified period of time.

4. Independence: two or more subjects can distribute or revoke policy attributes to the same subject for an object independent of each other.

5. Time-dependency: the effect of the distribution and revocation of policy attributes must take place at a certain time and must last for a specified period of time.

6. Location-dependency: the distribution and revocation of policy attributes must take place at a certain location.

The review rules determine which of the following two kinds of review are supported and impose conditions constraining the review of attributes.

1. Per-object review: for an object, find all (or a specified class of) attributes that govern the relationship between that object and a specified set of subjects that may directly or indirectly access that object.

2. Pr-subject review: for a subject, find all (or a specified class of) policy attributes which govern the relationship between that subject and a specified set of objects that subject may directly or indirectly access.

The imposed conditions for allowing the review of attributes determine, in particular, which users of an object may discover which users have access to that object, as well as what subjects may be used to access that object.

The coverage of attribute-review rules is specified in terms of the kinds of objects and subjects to which they apply. If different rules and conditions apply to different subjects and objects, the totality of these rules must be shown to support the defined policy objectives. If a composition of several policies is to be supported, attribute administration rules must be composed.

**A.3.1.4 Creation and Destruction of Subjects and Objects.** The rules for allowing the creation and destruction of subjects and objects must be defined. These rules impose the following conditions under which subjects and objects can be created and destroyed.

1. Creation and destruction authorization: the authorization of specific subjects to create and destroy a subject or an object and with what attributes.
2. Object reuse: the revocation of all authorizations to the information contained within a storage object prior to initial assignment, allocation or reallocation of that storage object to a subject from the TCB's pool of unused storage objects; no information, including encrypted representations of information, produced by a prior subject's actions should be available to any unauthorized subject.
3. Space availability: the capacity and presence of storage space shall be available for the creation of a subject or object.
4. Definition of default attributes subject or the default values and rules for inheriting object attributes, if any, shall be defined.

**A.3.1.5 Object Encapsulation** The encapsulation subcomponent of an access control policy specifies that a subject's access to a objects be constrained in such a way that (1) all accesses to these objects occur via access to a logically and/or physically isolated set of subjects that protect these objects from more general forms of access, with each subject having a unique protected entry point; and (2) confinement of this set of protecting subjects is such that these subjects cannot access any other objects and cannot give away access to the objects they protect.

Discretionary encapsulation allows individual (privileged and unprivileged) users to create protected subsystems and to set access to them at their own discretion (perhaps using well-known discretionary access control mechanisms). Nondiscretionary encapsulation uses logical and/or physical domains (and perhaps security levels) to enforce encapsulation at the product level (i.e., by system administrators as opposed to at the discretion of the creator of the protected subsystem). The traditional DoD mandatory policies may be useful for encapsulation in some environments. For example, one could use DoD mandatory policies to encapsulate a protected subsystem by reserving a sublattice of compartments for the programs and data objects of that subsystem. (Some trusted database management systems use this approach for the support of per-client Database Management System (DBMS) servers. The server(s) and database objects are encapsulated in a reserved sublattice of the TCB). Note that both discretionary and non-discretionary encapsulation can involve the use of surrogate subjects to protect the entry points to protected subsystems.

Rules for object encapsulation must be defined whenever object encapsulation is supported. Rules for object encapsulation constrain (1) access authorization to encapsulated objects (i.e., a subject access to an object can take place only if the subject invokes another subject that performs the requested action on the object using additional authorizations associated with the encapsulation); (2) application-level encapsulation (i.e., they define conditions for the creation of encapsulated subsystems); and (3) invocation of encapsulated subsystems.

#### A.3.1.6 Examples of Access Control Policy Characterization using the Proposed Framework

**A.3.1.6.1 Example 1: POSIX DAC** As a first example, we outline the characterization of the POSIX DAC within the proposed framework.

1. Definition of Subject and object policy attributes
    - subject attributes:
        - UID, list of GIDs, primary group
        - real and effective UID, GID
        - privilege vector
    - object attributes:
        - ACL contents (including ownership specifications)
2. Access Authorization
    - effective access algorithm rules (e.g., ACL search and mode determination)
    - privilege check rules of system calls
    - process isolation rules
3. Attribute Administration
    - distribution and revocation rules
        - owner based for permission; administrator based for privileges
        - selective on a user and group basis
        - transitive: not supported
        - immediate; not supported (e.g., a subject retains permissions to a copy of the accessed object until the subject dies; immediate at the object ACL level)
        - time dependent: not supported
        - location dependent: not supported
    - access review rules
        - per subject review: not supported
        - per object review: owner based (note that in old UNIX
        - versions any user could inspect the protection bits of a file regardless of whether he had any access to that file)

4. Creation and Destruction of Subjects and Objects

   - creation:
     - subjects - restricted to a fixed number (e.g., a fixed number of processes can be active per user at any time);
     - objects - unrestricted
   - default attribute determination
     - subjects - restricted to subject creator
     - objects - restricted by the umask and inheritance rules based on directory placement
   - destruction:
     - subjects - must be the creator of that subject
     - objects - must own that object
   - object reuse - supported for subjects and objects on object and extension (e.g., append operation) and creation.
   - space availability rules - not uniformly enforced.

5. Object Encapsulation

   - rules of the setuid/setgid functions in UNIX

**A.3.1.6.2 Example 2: MULTICS MAC** As a second example, we outline the MAC policy of MULTICS as defined by the Bell-LaPadula model.

1. Definition of Subject and object policy attributes

   - subject attributes:
     - subject maximum level
     - subject current level
     - ring of protection bracket (used in downward calls and upward returns)
   - object attributes:
     - object security level

2. Access Authorization

   - Bell-LaPadula model rules (i.e., *-property, simple security condition)
   - rules for ring check on downward call
   - process isolation rules

3. Attribute Administration

- distribution and revocation rules
  - rule for changing the subject security level (privileged)
- selective on a user and group basis
  - rule for changing the object security level (privileged)
  - transitive: not supported
  - immediate; supported
  - time dependent: not supported
  - location dependent: not supported
- access review rules
  - per subject review: only partially supported in the sense that the security levels of all objects accessible to a subject are known (via the Bell_LaPadula rules)
  - per object review: based on who controls access to the object

4. Creation and Destruction of Subjects and Objects

   - creation:
     - subjects - rule for current security level determination rule for process creation
     - objects - rules for user created objects including upgraded directories;
   - default attribute determination: not applicable
   - destruction:
     - subjects - must be the level of the subject
     - objects - must be at the level of the object or lower
   - object reuse - supported for subjects and objects on object and extension (e.g., append operation) and creation.
   - space availability rules - based on directory space quotas

5. Object Encapsulation: not supported

## A.4 Examples of Composing Access Control Policies

Many of the access control policies supported by a system represent a composition of two or more basic access control policies. The need to compose basic policies arises for at least two reasons. First, to extend the range of an system applicability, new applications subsystems or individual functions may be added to a TCB. These subsystems and functions may support different basic access control policies from those supported by the original TCB. These different policies must be composed with those of the original TCB. Second, to support new system or organizational policies, functions implementing new basic access policies are required to be added to a system's TCB. These new access control policies must also be composed with the existing ones to enable the implementation of the protection objectives of an organization.

The composition of access control policies within an access control component requires that both the individual access control policies and their rules for composition be completely defined (i.e., for each element of the defined policy, a corresponding set of rules must establish the completeness of the composition).

### A.4.1 Composition of Discretionary and Non-Discretionary Policies within the same TCB.

A typical example of access control policy composition within the same system's TCB is provided by the addition of a non- discretionary access control policy (e.g., the DoD mandatory policy) to a TCB that originally supports only a discretionary policy.

Suppose that the composition rules for the resulting TCB access control policy include the following specifications.

1. the composed policy attributes of each subject and object consist of the union of the DAC and MAC policy attributes;
2. the composed access authorization rules require that:
   (a) every action of a subject to an object represents the conjunction of both the respective DAC and MAC access checks; (i.e., both the mandatory and discretionary authorization rules are enforced on every subject and object protected by discretionary controls);
   (b) DAC authorization checks are themselves subject to the MAC authorization checks (i.e., the references issued by the enforcement modules of the discretionary policy are subject to the mediation specified by the mandatory rules).
3. the composed attribute administration rules represent the union of the DAC and MAC administration rules, and the DAC and MAC rules are enforced independently;
4. the composed subject and object creation and destruction rules require the conjunction of the object creation and destruction authorization rules; i.e., the creation and destruction of subjects and objects must pass both the DAC and MAC rules;
5. the composed object encapsulation rules represent the union of of the DAC and MAC encapsulation rules (if any).

To understand the subtlety of the interaction between the rules of the different policy areas consider the composed access authorization (i.e., conjunction and precedence of enforcement) and attribute administration rules (i.e., union and independent enforcement). The precedence of authorization enforcement is important whenever the exceptions returned by the enforcement of the two sets of rules are different. The reason is that if non-identical exceptions are returned by the two sets of rules, new covert channels may appear that would not appear had only the mandatory authorization rules be enforced. These covert channels would violate the intent of the mandatory secrecy policy.

**A.4.1.1 Example A1** Consider a system where a (secrecy or integrity) MAC policy is added to an existing DAC policy. The MAC authorization rules must be enforced by all TCB system calls. Assume that the exceptions returned when enforcing MAC authorization differ from (at least some of) those returned by DAC authorization. (Since exception returns must be sufficiently informative to allow a computation to recover from such an exception, we cannot assume, in practice, that all MAC and DAC exception returns are identical.) Since the MAC and DAC authorization exception returns may differ, the precedence of MAC and DAC authorization becomes access-control relevant in the composed policy.

Suppose that DAC authorization checks (exception: DACEX) take place before SECRECY MAC authorization checks (exception: MACEX). A subject at secrecy level H > L can use the DAC attribute administration rules, which are enforced independently of the corresponding MAC administration rules, to distribute/revoke access to a subject at secrecy level L for an object O at secrecy level H. By using these rules, the H subject can leak the contents of the object O to the L subject as follows: a bit of O equal to 1, the H subject denies access Read to the L subject for O, and for a bit of O equal to 0, the H subject allows access Read to the L subject for O. Thus, the L subject can decode the contents of object O at level H > L by decoding the string of exceptions it receives when it attempts to Read object O. For example, if the DAC authorization checks take place before MAC authorization checks, a string of exceptions DACEX, DACEX, DACEX, MACEX, MACEX, DACEX represents the bit string 111001. This is the case because DACEX represents the fact that the L subject is not allowed to Read object O, and MACEX represents the fact that the L subject is allowed to Read object O by the H subject, but the Read action is correctly rejected by the MAC authorization check. Should the precedence of the DAC/MAC authorization checks be such that the DAC authorization checks would always be themselves the subject to the MAC checks, this covert channel would not exist because all exception returns to the L subject would be MACEX, thereby conveying no information. Hence the precedence of the MAC and DAC authorization checks as specified in composition rule (2)(b) above matters.

A similar scenario of covert channel can be provided for INTEGRITY MAC functions and DAC functions. In that scenario a subject at a high integrity level takes illegal input from a subject at a low integrity level. (This relevance of covert channels to mandatory integrity policies is illustrated in the Covert-Channel Analysis Guideline [48].)

### A.4.2 Composition of Multiple Non-Discretionary Policies within the same TCB.

Examples of policy composition within the same TCB include those in which the DoD mandatory secrecy policy and a mandatory integrity policy are supported.

This composition might imply (1) that both the mandatory authorization rules be enforced on every subject and object reference and (2) that the attribute administration rules of the two mandatory policies must be compatible with each other. Compatibility of these rules would imply, for example, that the secrecy and integrity upgrade and downgrade conditions must not introduce covert channels that otherwise would not exist when the individual policies were used separately.

Suppose that the composition rules for the resulting TCB access control policy include the following specifications.

1. the composed policy attributes of each subject and object consist of the union of the secrecy and integrity policy attributes;

2. the composed access authorization rules require that every action of a subject to an object be conjunction of both the respective secrecy and integrity access checks (i.e., both types of authorization rules are enforced on the same set of subjects and objects), and neither rule takes precedence over the other;

3. the composed attribute administration rules represent the union of the secrecy and integrity administration rules; the two sets of administration rules are enforced independently;

4. the composed subject and object creation and destruction rules require the conjunction of the object creation and destruction authorization rules;

5. the composed object encapsulation rules represent the union of secrecy and integrity encapsulation rules (if any).

**A.4.2.1 Example A2** Consider a MAC secrecy policy defined by the Bell-LaPadula Multics interpretation with a restricted attribute administration rule Change-Object-Security-Level. Under this rule, a secrecy-unprivileged subject can only upgrade the secrecy level of an object; to downgrade the secrecy level of an object, the subject must be secrecy-privileged. Consequently, in this policy, the Change-Object-Security-Level rule can be used by secrecy-unprivileged subjects for upgrades and to secrecy-privileged subjects for downgrades. We will call the two options of this rule, the S-down and S-up rules. Similarly, consider a MAC integrity policy defined by the Biba strict integrity model augmented with an unprivileged downgrade rule for objects' integrity levels, called the I-down rule, and a privileged upgrade rule for objects' integrity, called the I-up rule. Assume that both these attribute administration policies are implemented within the same TCB. (This is not unrealistic. Both the SCOMP operating system and the GEMSOS TCBs compose MAC secrecy and integrity models.) In this example, we show that the composition of these two policies introduce both secrecy and integrity exposures that do not appear before composition.

Let P(SL,IL) denote a subject running at secrecy level SL and integrity level IL, and F (SL,IL) denote an object created at secrecy level SL and integrity level IL. In this example, SL (IL) is either Hs (Hi) or Ls (Li), where Hs > Ls (Hi > Li). Consider the following scenario (please see the attached figures)

(Note that in the Scenarios 1 - 4 below it is unimportant that the integrity and secrecy exceptions differ.)

Scenario 1: Let integrity-unprivileged subject P1(Hs,Hi) invoke the I-down rule to downgrade its object F(Ls,Hi) to F(Ls,Li). Secrecy-unprivileged subject P2(Ls,Hi), which could READ object F(Ls,Hi) before the downgrade, can no longer READ object F(Ls,Li), because the access-authorization rule of the MAC integrity policy rejects that action. This means that secrecy-unprivileged subject P2(Ls,Hi) can tell whether integrity-unprivileged subject P1(Hs,Hi) invoked the I-down rule. This is a covert channel that did not exist before the MAC secrecy and integrity policies were composed. This channel represents a violation of the MAC secrecy policy because an unprivileged subject at a low secrecy level can sense a change of behavior of an integrity-unprivileged subject at a high secrecy level. (A Trojan Horse in the high secrecy subject can produce illegal output for a low secrecy subject.)

Now let us consider Scenario 2, which is the "dual" of the Scenario 1.

Scenario 2: Let secrecy-unprivileged subject P2(Ls,Li) invoke the S-up rule to upgrade its object F (Ls, Hi) to F (Hs, Hi). Integrity-unprivileged subject P1(Ls,Hi), which could READ object F(Ls,Hi) before the upgrade, can no longer READ object F (Hs, Hi), because the access authorization rule of the MAC secrecy policy rejects that action. This means that integrity-unprivileged subject P1(Ls,Hi) can tell whether secrecy-unprivileged subject P2(Ls,Li) invoked the S-up rule. This is a covert channel that did not exist before the MAC secrecy and integrity policies were composed. This channel represents a violation of the MAC integrity policy because an unprivileged subject at a high integrity level can sense a change of behavior of an secrecy-unprivileged subject at a low integrity level. (A Trojan Horse in the high integrity subject can obtain illegal input from a low integrity subject.)

The two scenarios above show that (1) the S-up rule of the secrecy policy is incompatible with the access-authorization rules of the integrity policy, and that (2) the I-down rule of the integrity policy is incompatible with the access-authorization rules of the secrecy policy. Thus, the secrecy and integrity policies cannot be composed as specified.

Two similar scenarios illustrate the same types of covert channels whenever two privileged downgrade/upgrade rules, S-down and I-up, (which are also restricted versions of the Change-Object- Security-Level of the Bell-LaPadula Multics interpretation and Biba models) are considered. The covert channels obtained are NO less interesting than the previous ones, despite the use of privileged attribute-administration rules, because (1) the secrecy covert channel is created by an integrity (NOT secrecy) privileged subject, and (2) the integrity covert channel is created by a secrecy (NOT integrity) privileged subject. Thus, the to subjects are unprivileged with respect to the types of exposures they can cause. These, scenarios also illustrate the unexpected effects of composing two sound policies.

Consider the Scenarios 3 and 4 below.

Scenario 3: Let the secrecy-unprivileged, but integrity-privileged, subject P1(Hs,Li) invoke the I-up rule to upgrade its object F(Hs,Li) to F(Hs,Hi). Secrecy-unprivileged subject P2(Ls,Li), which could WRITE (APPEND TO) the object F(Hs,Li) before the upgrade, can no longer WRITE (APPEND TO) the object F(Hs,Hi), because the access-authorization rules of the MAC integrity policy rejects that action. This means that secrecy-unprivileged subject P2(Ls,Li) can tell whether secrecy-unprivileged, but integrity-privileged, subject P1(Hs,Li) invoked the I-up function. This is a covert channel that did not exist before the MAC secrecy and integrity policies were composed. This channel represents a violation of the MAC secrecy policy because a secrecy-unprivileged subject at a low secrecy level can sense a change of behavior of a secrecy-unprivileged subject at a high secrecy level. (A Trojan Horse in the high secrecy subject can produce illegal output for a low secrecy subject.)

Now let us consider Scenario 4, which is the "dual" of the Scenario 3.

Scenario 4: Let the integrity-unprivileged, but secrecy-privileged, subject P2(Hs,Li) invoke the S-down function to downgrade its object F(Hs,Li) to F(Ls,Li). Integrity-unprivileged subject Pl(Hs,Hi), which could WRITE (APPEND TO) object F(Hs,Li) before the downgrade, can no longer WRITE (APPEND TO) object F(Ls,Li), because the access-authorization rule of the MAC secrecy policy rejects that action. This means that integrity-unprivileged subject Pl(Hs,Hi) can tell whether secrecy-privileged, but integrity-unprivileged, subject P2(Hs,Li) invoked the S-down function. This is a covert channel that did not exist before the MAC secrecy and integrity policies were composed. This channel represents a violation of the MAC integrity policy because an integrity-unprivileged subject at a high integrity level can sense a change of behavior of an integrity-unprivileged process at a low integrity level. (A Trojan Horse in the high integrity subject can obtain illegal input from a low integrity subject.)

### A.4.3 Composition of Trusted Subsystems (i.e., Servers) within a TCB

A typical example of composition appears when a subsystem implementing its own access control policy (i.e., a trusted server) is integrated within an operating system TCB. (An alternate way of integrating such a subsystem in a trusted operating system is illustrated in the following discussion of TCB policy subsetting). Such subsystem integration is fairly common of database management systems and operating systems. Since these subsystems implement their own policies, which generally differ from those of the operating system, the composition must ensure that neither the operating system nor the subsystem interfaces of the same TCB would allow an untrusted application or an unprivileged user to access (1) operating system objects in an unauthorized manner via the subsystem (i.e., trusted server) interface, or (2) subsystem objects via the operating system interface in an unauthorized manner. Furthermore, when non- discretionary access controls are implemented in both the underlying operating system TCB and the subsystem, the composition of the two should not introduce covert channels that were not present when the individual policies were supported.

The suggested composition causes the access control partitioning of the TCB into an operating system and a subsystem partition (not to be confused with a policy partition, which is discussed below). The two partitions can share other TCB policy components such as identification and authentication, system entry, and trusted path. Other similar examples of system-object partitioning are offered by message or mail subsystems and communication protocol subsystems.

### A.4.4 Composition by Policy Subsetting

An alternate method of policy composition is that provided by policy subsetting. In this method, separate TCB subsets are allocated different policies. This method of policy composition is addressed in detail in the Trusted Database Management System Interpretation of the Trusted Computer System Evaluation Criteria [11].

In this composition method a TCB subset, M, is a set of software, firmware, and hardware (where any of these three could be absent) that mediates the access of a set of subjects, S, to a set of objects, O, on the basis of a stated access control policy, P, and satisfies the properties or the reference validation mechanism [11]. M uses resources provided by an explicit set of more primitive TCB subsets to create the objects of O, create and manage its data structures, and enforce the policy P. (The above definition does not explicitly prohibit an access control policy P that allows trusted subjects.) If there are no TCB subsets more primitive than M, then M uses only hardware resources to instantiate its objects, to create and manage its own data structures, and to enforce its policy. However, if M is not the most primitive TCB subset, then M does not necessarily use the hardware or firmware functions to protect itself. Rather, it uses either hardware resources or the resources provided by other, more primitive TCB subsets. Thus TCB subsets build on abstract machines, either physical hardware machines or other TCB subsets. Just like reference validation mechanisms, a TCB subset must enforce a defined access control policy separately than those enforced by other subsets.

The access control policy P[i] is the policy allocation for each identified TCB subset M[i] of a product along with the relation of these policies to the product policy P. The allocated policies P[i] will be expressed in terms of subjects in S[i] and objects in O[i]. To satisfy the requirement that the (composite) TCB enforce its stated policy P, each rule in P must be traceable through the structure of the candidate TCB subsets to the TCB subset(s) where that enforcement occurs. It must also be noted that every subject trusted with respect to P[i] must be within the TCB subset M[i].

**A.4.4.1 Composition by Policy Partitioning** In most distributed systems and networks, security policies are supported by different system components, some of which can be independently developed products. The partitioning reflects different architectural goals including support for client-server architectures, single-login, and ease of administration. In some distributed systems and networks, the access control policy is partitioned across several system components, also reflecting a variety of architectural goals. For example, in some distributed systems, the definition of subject access control attributes is performed by a separate security server (e.g., the privileged server in OSF's DCE) whereas the access authorization is performed autonomously by each application server. In other systems, even the access authorization is performed across several components reflecting the partitioning of subject and object implementation. For example, in some networked systems, access authorization checks for file system objects is performed by every server whereas authorization checks for communication objects is performed by the communication system (e.g., on the end-to-end session and connection objects).

The access control policy allocation to different components require the specification of the overall composite trusted system policy as well as the determination that each policy partition supports that policy. In the proposed framework, one can specify the policy areas and rules supported by each partition and the composition of partition policies. Thus, one can define the overall trusted system policy and provide the argument that the individual partition policies support the overall policy.

### A.4.5 Conclusions and Completeness Arguments

We presented a framework for security policy characterization consisting of guidelines for (1) identification and authentication, (2) access control, and (3) audit policy definition. we illustrated this framework with specific policy examples and requirements. The question as to whether the proposed framework for security policy specification is complete arises naturally. For example, can all possible access control policies be specified within the proposed framework ? Although no formal theory can answer this question to date, there are three informal reasons for which we believe that the answer to this question is affirmative.

First, all formal models of, and criteria requirements for, access control policies include a subset of the five areas proposed by our framework. Thus, our framework represents a union of all policy areas addressed by all formal models and criteria proposed to date.

Second, the types of rules listed in each area appear to cover all types of rules defined in all formal models, and requirements in all security criteria, proposed to date.

Third, review of access control functions and mechanisms published in the literature to date shows that they implement policies describable within the proposed framework.

For the above three reasons we believe that the proposed framework of access control is complete. For similar reasons we believe that the identification and authentication, and audit frameworks are also complete.

## References

[1] J. P. O'Connor. Trusted RUBIX: A multilevel secure client-server DBMS. In *Proceedings of the Eigth Annual IFIP Working Group 11.3 Working Conference on Database Security*, August 1994.

[2] Infosystems Technology, Inc. *Trusted RUBIX Version 2.0 User's Guide/Tutorial*, March 1994.

[3] Infosystems Technology, Inc. *Trusted RUBIX Version 2.0 Reference Guide*, March 1994.

[4] Infosystems Technology, Inc. *Trusted RUBIX Version 2.0 Trusted Facility Manual*, March 1994.

[5] Infosystems Technology, Inc. *Trusted RUBIX Version 2.0 Security Features User's Guide*, March 1994.

[6] Infosystems Technology, Inc. Software/segment specification for the Trusted Distributed RUBIX system. Technical Report TR-9401-00-01, Infosystems Technology, Inc., December 1994.

[7] Infosystems Technology, Inc. Software requirements specification for the Trusted Distributed RUBIX System. Technical Report TR-9402-00-01, Infosystems Technology, Inc., December 1994.

[8] Infosystems Technology, Inc. Software design document for the Trusted Distributed RUBIX system. Technical Report TR-9501-00-00, Infosystems Technology, Inc., February 1995.

[9] Infosystems Technology, Inc. Interface design document for the Trusted Distributed RUBIX system. Technical Report TR-9502-00-00, Infosystems Technology, Inc., February 1995.

[10] W. Shockley and R. R. Schell. TCB subsets for incremental evaluation. *Proceedings of the Third Aerospace Computer Security Conference*, December 1987.

[11] National Computer Security Center. Trusted database interpretation of the trusted computer system evaluation criteria. Technical Report NCSC-TG-021, National Computer Security Center, April 1991.

[12] Department of Defense. Department of Defense trusted computer system evaluation criteria. DOD Standard 5200.28-STD, Department of Defense, December 1985.

[13] X/Open. Data Management: SQL Call Level Interface (CLI). X/OPEN Preliminary Specification, X/Open, October 1993.

[14] T. F. Keefe and W. T. Tsai. Multilevel concurrency control for multilevel secure database systems. In *Proceedings of the 1990 Symposium on Security and Privacy*, May 1990.

[15] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Francisco, California, 1993.

[16] X/OPEN. Distributed transaction processing: Reference model. X/OPEN guide, X/OPEN, 1993.

[17] X/OPEN. Distributed transaction processing: The tx (transaction demarcation) specification. X/OPEN Preliminary Specification, X/OPEN, October 1992.

[18] X/OPEN. Distributed transaction processing: The xa specification. X/OPEN CAE specification, X/OPEN, December 1991.

[19] S. Jajodia and C. McCollum. Using 2-phase commit for crash recovery in federated multilevel secure database management systems. In *Proceedings of the 3th IFIP Working Conference on Dependable Computing for Critical Applications*, Mondello, Italy, September 1992.

[20] LouAnna Notargiacomo. MUSET (MUltilevel SEcure (MLS) Transaction) Distributed DBMS. In *Proceedings of the 1994 Rome Laboratory Workshop on Database Security*, July 1994.

[21] T. Lunt, D. Denning, R. Schell, M. Heckman, and W. Shockley. Element-level classification with A1 assurance. *Computers and Society*, August 1988.

[22] Xiaolei Qian and Teresa Lunt. Semantic interoperation: A query mediation approach. Technical Report SRI-CSL-94-02, SRI International, Computer Science Laboratory, April 1994.

[23] International Organization for Standardization. Remote database access – part 1: Generic model, service, and protocol. Draft International Standard ISO/IEC DIS 9579-1, International Organization for Standardization, 1991.

[24] International Organization for Standardization. Remote database access – part 2: Sql specialization. Draft International Standard ISO/IEC DIS 9579-2, International Organization for Standardization, 1991.

[25] X/OPEN. ACSE/Presentation: Transaction Processing API (XAP-TP). X/OPEN Preliminary Specification, X/OPEN, January 1994.

[26] X/OPEN. Structured query language (SQL). X/OPEN CAE specification, X/OPEN, August 1992.

[27] Infosystems Technology, Inc. Trusted RUBIX Formal Model of Access Control. Technical Report TR-9504-00-02, Infosystems Technology, Inc., November 1995.

[28] T. Lunt, D. Denning, R. Schell, W. Shockley, and M. Heckman. The SeaView security model. *IEEE Transactions on Software Engineering*, June 1990.

[29] T. Lunt and D. Hsieh. The SeaView secure database system: A progress report. In *Proceedings of the European Symposium on Research on Computer Security*, Toulouse, France, October 1990.

[30] Infosystems Technology, Inc. TCB subset DBMS architecture project: Design document. Technical Report TR-9403-00-01, Infosystems Technology, Inc., December 1994.

[31] Infosystems Technology, Inc. TCB subset DBMS architecture project: Final report. Technical Report TR-9404-00-01, Infosystems Technology, Inc., December 1994.

[32] R. Bayer and M. Schkolnick. Concurrency of operations on b-trees. *Acta Informatica*, 9(1):1–21, 1977.

[33] P. Lehman and B. Yao. Efficient locking for concurrent operations on b-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[34] Arca Systems, Inc. Assured DAC for trusted RDBMSs: Final report. Technical Report ATR94020, Arca Systems, Inc., September 1994.

[35] D. Sterne. On the buzzword security policy. *Proceedings of the 1991 Symposium on Security and Privacy*, pages 219–230, May 1991.

[36] J. P. O'Connor. Applying the concept of TCB subsets to trusted subject DBMS architectures. In *Proceedings of the 1994 Rome Laboratory Workshop on Database Security*, July 1994.

[37] National Computer Security Center. Trusted network interpretation of the trusted computer system evaluation criteria. Technical Report NCSC-TG-005, National Computer Security Center, July 1987.

[38] D. McCullough. Specifications for multilevel security and a hook-up property. In *Proc. of the 1987 IEEE Symposium on Security and Privacy*, pages 161–166, Oakland, California, April 1987.

[39] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. of the 1982 IEEE Symposium on Security and Privacy*, pages 11–20, Oakland, California, April 1982.

[40] J. A. Goguen and J. Meseguer. Unwinding and inference control. In *Proc. of the 1984 IEEE Symposium on Security and Privacy*, pages 75–85, Oakland, California, April 1984.

[41] J. Millen. Hookup security for synchronous machines. In *Proc. of Computer Security Foundations Workshop*, pages 84–90, Franconia, New Hampshire, June 1990.

[42] D. McCullough. Noninterference and composability of security properties. In *Proc. of the 1988 IEEE Symposium on Security and Privacy*, pages 177–186, Oakland, California, April 1988.

[43] D. Johnson and E.J. Thyer. Security and the composition of machines. In *Proc. of the Computer security Foundations Workshop*, pages 72–89, Franconia, N.H., June 1988.

[44] J. Millen. Unwinding forward correctability. In *Proc. of Computer Security Foundations Workshop*, pages 2–10, Franconia, New Hampshire, June 1994.

[45] E. S. Lee, P.I.P. Boulton, B.W. Thompson, and R.E Soper. Composable trusted systems. Technical report, University of Toronto, Computer Systems Research Institute, Toronto, Ontario, Canada, May 1992.

[46] H. M. Hinton and E.S. Lee. The compatibility of policies. In *Proc. 1994 ACM Conference on Computer and Communications Security*, pages 258–269, Fairfax, Virginia, November 1994.

[47] M. Abadi and L. Lamport. Composing specifications. Technical Report 66, DEC Systems Research Center, Palo Alto, California, October 1990.

[48] National Computer Security Center. A Guide to Understanding Covert Channel Analysis of Trusted Systems. Technical Report NCSC-TG-030, National Computer Security Center, November 1993.

☆U.S. GOVERNMENT PRINTING OFFICE: 1997/509-127/47085

## *MISSION*
## *OF*
## *ROME LABORATORY*

Mission. The mission of Rome Laboratory is to advance the science and technologies of command, control, communications and intelligence and to transition them into systems to meet customer needs. To achieve this, Rome Lab:

a. Conducts vigorous research, development and test programs in all applicable technologies;

b. Transitions technology to current and future systems to improve operational capability, readiness, and supportability;

c. Provides a full range of technical support to Air Force Materiel Command product centers and other Air Force organizations;

d. Promotes transfer of technology to the private sector;

e. Maintains leading edge technological expertise in the areas of surveillance, communications, command and control, intelligence, reliability science, electro-magnetic technology, photonics, signal processing, and computational science.

The thrust areas of technical competence include: Surveillance, Communications, Command and Control, Intelligence, Signal Processing, Computer Science and Technology, Electromagnetic Technology, Photonics and Reliability Sciences.